Titre: Title:	Verifying Timed LTL Properties Using Simulink Design Verifier
Auteur: Author:	Mohammad-Reza Gholami
Date:	2016
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Gholami, M.-R. (2016). Verifying Timed LTL Properties Using Simulink Design Verifier [Thèse de doctorat, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/2120/

URL de PolyPublie: PolyPublie URL:	https://publications.polymtl.ca/2120/
Directeurs de recherche: Advisors:	Hanifa Boucheneb
Programme: Program:	Génie informatique

UNIVERSITÉ DE MONTRÉAL

VERIFYING TIMED LTL PROPERTIES USING SIMULINK DESIGN VERIFIER

MOHAMMAD-REZA GHOLAMI
DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

THÈSE PRÉSENTÉE EN VUE DE L'OBTENTION
DU DIPLÔME DE PHILOSOPHIÆ DOCTOR
(GÉNIE INFORMATIQUE)
AVRIL 2016

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Cette thèse intitulée :

VERIFYING TIMED LTL PROPERTIES USING SIMULINK DESIGN VERIFIER

présentée par : GHOLAMI Mohammad-Reza
en vue de l'obtention du diplôme de : Philosophiæ Doctor
a été dûment acceptée par le jury d'examen constitué de :

Mme BELLAÏCHE Martine, Ph.D., présidente
Mme BOUCHENEB Hanifa, Doctorat, membre et directrice de recherche
Mme NICOLESCU Gabriela, Doctorat, membre
M. BENTAHAR Jamal, Ph.D., membre

DEDICATION

This thesis is dedicated with love to my wife, Baharafarin, and two beautiful children, Daniel and Diana.

ACKNOWLEDGEMENTS

There are so many important people to whom I owe my sincere gratitude, and I will try to highlight some of them in these acknowledgements. First off, I would like to thank my supervisor, Professor Hanifa Boucheneb, for encouraging me and supporting me without hesitation throughout my studies. Her dedication to my research and unwavering guidance has been incredibly helpful and inspiring – words cannot express my appreciation for Dr. Boucheneb's efforts.

While the destination is grand, I would be nowhere without the journey to get here. This journey with my colleagues at Ecole Polytechnique de Montreal has provided me with lifetime experiences, and has set me up for a prosperous future. I would like to thank the Department of Computer and Software Engineering for affording me the opportunity to learn and grow with the best.

I would like to thank the committee members, Professor Martine Bellaiche, Professor Gabriela Nicolescu, and Professor Jamal Bentahar, for their attention and dedication of their time to evaluate my research work. I would like to thank my colleagues at VeriForm Laboratory, especially Parisa Heidari and Aurel Randolph. I would also like to thank Daniel Wolfe, my manager at IsaiX Technologies Inc. for his support throughout this process.

Lastly, but certainly not least, I would like to thank my "inner circle" support system : my family. My amazing wife, Baharafarin, for her incredible patience and huge sacrifice – this has been a tiresome journey for her as much as it has been for me. My mother-in-law and father-in-law, Maryam RezaiTabar and Hossein Esbati, who have experienced this journey with my wife and I, and have been so helpful in raising our children and taking care of our household. My parents, for always encouraging me to be the best I can be, and pursue the highest level of education. And of course, my children, Daniel and Diana, for being understanding when their "Baba" didn't have time to play, and had to work on his thesis.

RÉSUMÉ

Les logiciels jouent un rôle de plus en plus important dans les systèmes embarqués notamment dans les domaines de la santé, de l'automobile et de l'avionique. Un objectif important du génie logiciel est d'offrir, aux développeurs, un support ainsi que les outils d'aide à la conception de systèmes fiables nonobstant leur complexité.

Dans le but d'atteindre cet objectif, des environnements de développement comme Simulink et SCADE proposent un processus de développement, basé sur des modèles, qui intègre, d'une manière réfléchie, différentes approches et outils de vérification (test, simulation, vérification formelle, évaluation, génération de code, etc). Ils permettent ainsi de concevoir, tester, simuler, vérifier, corriger des modèles puis de générer automatiquement du code à partir de ces modèles.

Cette thèse s'intéresse aux méthodes formelles et à l'intégration de celles-ci dans l'environnement de développement Simulink. Les méthodes formelles s'appuient sur des outils mathématiques pour spécifier, par des modèles, le comportement et les propriétés d'un système et prouver qu'il satisfait ses requis. Simulink-Design-Verifier (SLDV) est un outil de vérification formelle, intégré à l'environnement de développement Simulink, qui permet de vérifier des propriétés de sûreté (assertions) sur des modèles Simulink. Cette thèse vise à étendre cette classe de propriétés à des propriétés linéaires LTL (Linear Temporal Logic), LTL temporisé et LTL à base d'événements. Les contributions de cette thèse sont présentées sous forme de trois articles.

Le premier article présente une étude de cas qui a permis d'expérimenter l'environnement de développement Simulink, d'identifier ses caractéristiques et ses limitations. Il s'agit de modéliser et vérifier un dispositif médical appelé sonde d'intubation. Une sonde d'intubation est une tubulure mise en place sur un sujet inconscient qui permet notamment d'assurer en permanence le passage de l'air vers les poumons. Ce système est composé de deux ballonnets, deux robinets d'accès pour gonflage manuel, deux capteurs de pression, un distributeur de puissance, une pompe et un réservoir d'air. Tous ces composants sont concurrents et contrôlés par contrôleur programmable décrit par un grafcet. Cet article montre comment utiliser l'environnement Simulink pour, d'une part, modéliser ces différents composants ainsi que leurs interactions, et d'autre part, vérifier formellement des propriétés, afin de s'assurer du bon fonctionnement du système. Cependant, la spécification de certaines propriétés temporelles n'est pas évidente car elles doivent être exprimées sous forme d'assertions. Les articles suivants proposent des blocks canevas pour des propriétés temporelles linéaires.

Le deuxième article est une version améliorée et étendue du premier article. Il s'est intéressé à réduire la complexité de vérification en modifiant significative le modèle et en proposant des blocks de spécification de propriétés linéaires basées sur les événements émis par le contrôleur.

Le troisième article est dédié à la spécification de propriétés LTL en utilisant SLDV. Il propose des blocs Simulink configurables qui spécifient ces propriétés. Le but de ces blocs est de transformer les propriétés en assertions qui sont vérifiables par SLDV.

La solution proposée dans le seconde et troisième article, est donc une extension de la bibliothèque de blocs de Simulink qui permet aux utilisateurs moins experts de spécifier et vérifier certaines propriétés LTL.

Ce travail est donc limité aux propriétés LTL à temps discret, et restreint à certaines propriétés LTL. Nos travaux futurs consisteraient à l'extension de la bibliothèque de blocs de Simulink pour supporter des propriétés LTL plus complexes et à plus grande échelle.

ABSTRACT

Software plays increasingly a significant role in embedded systems particularly used in healthcare, automotive and avionics. An important goal of software engineering is to offer developers support tools to design reliable systems despite the system complexity.

In order to achieve this, development environments like Simulink and SCADE propose a model-based development process, which integrates in a thoughtful way, different approaches and verification tools (test, simulation, formal verification, evaluation, code generation, etc.). They allow to design, test, simulate, verify, correct the models and then automatically generate code from these models.

This thesis is interested in formal methods and integrating them in the Simulink development environment. Formal methods are based on mathematical tools to specify the behavior and properties of a system by models, and prove, if it meets its requirements. Simulink Design Verifier (SLDV) is a formal verification tool, integrated in Simulink development environment, to verify safety properties (assertions) on Simulink models. This thesis aims to extend this class of properties to linear properties LTL (Linear Temporal Logic), timed LTL and event based LTL. The contributions of this thesis are presented in three articles.

The first article presents a case study that experiment the Simulink development environment, to identify its characteristics and limitations. It consists of modeling and verifying a medical device called intubation tube. An intubation tube is a tube that assures permanent air flow to the lungs of unconscious person. This system consists of two balloons, two access valves for manual inflation, two pressure sensors, a power distributor, a pump and an air reservoir. All these components work in parallel and are controlled by a programmable controller described by grafcet. This article shows how to use the Simulink environment, to model these components and also how to verify formally the properties to ensure the system is well functioning. However, the specification of certain temporal properties is not obvious because they must be expressed as assertions. The following articles propose canvas blocks for linear temporal properties.

The second article is an improved and extended version of the first article. It is interested in reducing verification complexity by changing significantly the model, and proposing specification blocks of linear properties, based on events issued by the controller.

The third article is dedicated to the specification of LTL properties using SLDV. It proposes configurable Simulink blocks that specify these properties. The purpose of these blocks is to

transform the properties into assertions that are verifiable by SLDV.

The solution proposed in the second and third articles, is to extend the block library of Similink, which allows less-expert users to specify and verify some Linear Temporal Logic (LTL) properties.

This work is limited to discrete time LTL properties, and restricted to specify some LTL properties. Our future work is devoted to extend the block library of Simulink to have support for a large scale and more complex LTL properties.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SIGNS AND ABREVIATIONS

ABS	Antilock Break System
BDD	Binary Decision Diagram
BMC	Bounded Model Checking
CTL	Computation Tree Logic
DAL	Design Assurance Level
FCA	Formal Concept Analysis
FM	Formal Methods
FMTS	Formal Methods Technical Supplement
FSM	Finite-State Machine
FV	Formal Verification
HLR	High Level Requirements
IEEE	Institute of Electrical and Electronics Engineers
LLR	Low Level Requirements
LTL	Linear Temporal Logic
MAAB	MathWorks Automotive Advisory Board
MC/DC	Modified Condition/Decision Coverage
MBD	Model-based Design
MDD	Model Driven Development
NASA	National Aeronautics and Space Administration
OBDD	Ordered Binary Decision Diagrams
PIL	processor-in-the-loop
PVS	Prototype Verification System
ROBDD	Reduced Ordered Binary Decision Diagrams
SAT	Satisfiability
SC	Software Consideration
SCADE	Safety-Critical Application Development Environment
SDLC	Software Development Life Cycle
SDP	Software Development Process
SG	Sub Group
SIL	software-in-the-loop
SLDV	Simulink Design Verifier
SMT	Satisfiability Modulo Theories
SMV	Symbolic Model Verifier

SSM	Safe State Machine
SysML	System Modelling Language
TAF	Test Automation Framework
TCG	Test Case Generation
TDD	Task Design Document
UML	Unified Modelling Language

LIST OF APPENDIX

CHAPTER 1 INTRODUCTION

Software plays an important role in almost every part of our everyday life from working in the office and driving the car to navigation systems in the aircraft. Over the last decades, reliability of complex software and hardware systems became increasingly crucial. Consequently, software verification of such systems, has been a significant problem in computer science for several years. Software failure could possibly cause catastrophic consequences regarding human life. In addition, a system is identified as *safety-critical*, when its failure causes catastrophic consequences, such as compromising the human life or even destruction to the system itself. Moreover, in the safety-related real-time system, verification and validation activities are becoming larger and more costly whenever its size and complexity grows.

A key objectives of software engineering is to support developers building a system that functions reliably even if it is complex. In order to attain this goal, formal methods are advised to be employed. To clarify, these methods are mathematically based techniques, and each tool has it own supported language. The use of formal methods has been addressed in DO-178B [1] and can significantly increase the knowledge of developers about the system [43]. In addition, formal methods disclose inconsistencies as well as ambiguities in the early phase of system design phase. As a result, they can be eliminated in order to make sure that the system has a suitable behavior.

Applying Model-Based Design [33] in safety-related applications and using formal verification, illustrates that the system fulfils its correctness criteria. For performing formal verification, the first need is to specify a formal model of the system. In addition, availability of executable models to perform verification, validation and test is one of the most effective factors of model-based design that helps one to apply formal verification techniques.

1.1 Definitions and Basic Concepts

In this section, we introduce some concepts related to the software development process and its enhancements. These concepts are the base and will be used in the process of applying formal method techniques into development life-cycle of software for the safety-critical embedded systems.

1. DO-178B is a standard for certification of software used in airborne systems

1.1.1 Software Development Process

The Software Development Process (SDP) sometimes is mentioned as Software Development Life Cycle (SDLC). In fact, it is a structure which is required for the development of software product. Moreover, it is a step-by-step process involved in the development of a software product as well as altering software systems, methodologies and models which people employ for developing these systems. Over the past years, efforts to enhance SDLC practices have been presented for improving the quality of software, reliability, and fault-tolerance. In addition, various software life cycle models exist which they describe different phases of the software development cycle as well as the order in which those phases are implemented. Furthermore, some companies have their own, but all models have very similar patterns which are just customized in purpose. Each model has its advantages and disadvantages, and it is up to the company's development team to use the most suitable one for the project.

In the following, we briefly describe some process models that are used for developing the safety-critical systems.

V-Shaped Model

The typical process of developing a safety-critical system is based on the V-Shaped life-cycle model [14]. This model is highly disciplined and it is also an extended version of the waterfall model which processes are executed in a sequential path. In addition, it is based on association of a testing phase for each matching development phase. Moreover, a system test plan is produced before the development phase is started. In this model, each phase in the cycle has a particular deliverables and should be verified in the same phase. In other words, the next phase of the development cycle starts only if the previous phase is successfully completed. Figure 1.1, illustrated the V-Shaped model.

Formal Methods Model

The formal methods model includes a set of activities that advances to the formal mathematical specification of software system [62]. If formal methods are employed during the software development, many of problems that are hard to find can be identified and eliminated by its provided mechanism. In particular, mathematical analysis helps easily to discover Ambiguity, inconsistency and incompleteness in the software system, so that they can be corrected early in the development. In other words, by using formal methods during the design phase, they assist as a basis for software verification, and consequently allows developers to identify and fix errors that might be otherwise go undetected. Several notations are available for

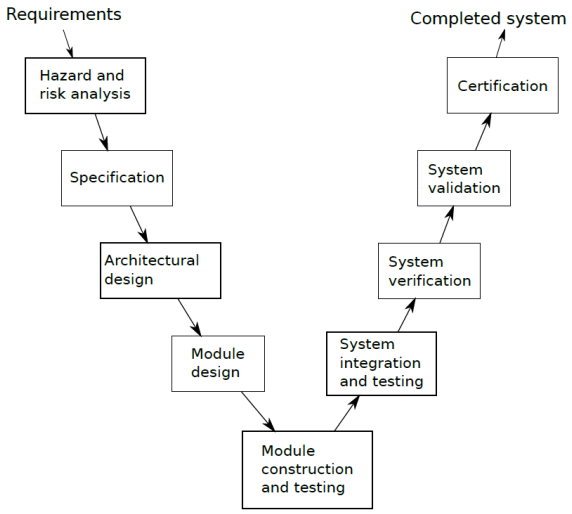


Figure 1.1 V-Shaped Life Cycle Model [69]

formalizing the software specifications. In general, the automata theory is the one that can be employed to design a system in finite state machines, in order to build and validate its behavior. When the software is used in the safety-critical system like avionics systems, it is recommended that formal methods be applied. Standards for safety assurance of software, like DO178B [43] request formal methods at the highest level of classification (Level A).

1.2 Problem Statement

The growth in sophistication in embedded systems used in safety-critical applications such as automotive, healthcare and avionics, necessitates to use more efficient processes for design and development. Ordinary design processes are not responsive enough in identifying the flaws in requirements; thus, the whole process would be more costly and take longer.

Even a sole error in the source code may cause the failure of the system. In a safety critical system, this failure can impose high costs and might even have the risk for the health of people. Imagine the Antilock Break System (ABS) in the car, navigation systems in an air plane or medical devices in the hospital operation room. For such safety-critical systems, availability of robust guidelines for software, have remarkable value for developers of these systems. In fact, guidelines should outline processes, artifacts and goals for the development of software that focuses on the quality.

In particular, support to build a system which functions reliably irrespective of its complexity, is a key objective of software engineering. Moreover, run-time errors that are not identified during the development phase of the software, might be serious. In addition, all commercial

software which are designed and developed for avionics in the U.S., are requested to comply with the DO-178 standard. The guidelines defined in DO-178 are projected to determine that developers of avionics systems employ a particular degree of process rigor. In fact, development and verification must follow a rigorous process to ensure that the software will perform its anticipated function with a suitable level of confidence in safety.

One of the ways to succeed this goal, is applying formal methods techniques, and tools to the software development life-cycle for specifying and verifying properties in such systems. Moreover, different levels of rigor or formality can be adopted, and different formal methods techniques can be used as well, to address the tasks in software development.

Although there are several cases proving the applicability of formal methods in the high complexity domains as well as industrial applications, they use more complex notations than other lightweight and intuitive graphical notations (e.g., Unified Modeling Language (UML)). So, creating a mathematical model of a system as well as specifying the functional requirements are not easy and could be mentioned as one of the important concerns to software producers. In addition, model checking technique might cause the state space explosion for large or complex systems. Moreover, the absence of easy to use tools that can assist the developers during the life cycle activities, could be another case. In addition to this, formal verification requires specific environment. Matlab/Simulink is an often used industrial tool in designing embedded systems. One of the primary uses of Matlab/Simulink is modeling the embedded software and its physical environment in a common formalism. This feature of the tool renders it highly valuable in the validation of embedded software design, leveraging numerical simulation. Having claimed this, formal verification of such models still proves problematic, as Simulink is a programming language without enough documented formal semantics. Furthermore, currently verification tools which have supports for DO-178C standard (e.g., SCADE Design Verifier and Simulink Design Verifier), does not support formal verification of continuous time systems.

1.3 Research Objectives

According to previously mentioned problems and obstacles, tremendous amount of effort is needed in order to have complete system verification acquiring the highest level of safety. In this chapter, we explain our objectives as well as the methodology to achieve those objectives. One of the main goals of this thesis is to address the essential problems for early detection of errors, in complex and safety critical applications, whereby facilitate the research in this area for developing and maintain a high-integrity application which comply with DO-178 [43] objectives for certification.

We summarize the detailed objectives of our research as the following items:

— To improve verification of critical systems we use Simulink and we add some customizable blocks into the Simulink library and make the formalization of the requirement specifications easier.
— Introducing a tool that facilitates instrumenting Simulink models by automatically adding some predefined and customizable properties to the model.
— Specifying some critical properties that are not easy to become formalized in Simulink.
— Investigation of applying verification for some LTL properties in Simulink Design Verifier.

1.4 Thesis Structure

The rest of this document is organized as follows: Chapter 2 presents a critical review on methods and environments for verification of safety critical systems. In particular, in this chapter, we review verification techniques and some available tools, and introduce Simulink Design Verifier which is qualified for development of safety-critical systems.

The next three chapters are articles that deal with formalization and verification of some safety and (timed) linear properties that are modeled in the Matlab/Simulink environment.

The first article, whose title is *Applying Formal Methods into Safety-Critical Health Applications*, is dedicated to the result of our studies for modeling and formalizing the requirements of an embedded system in the Simulink and Stateflow environment. We bring an introduction about the tool, its library of blocks and Stateflow as well. This article is the subject of the chapter 3, and is published in *Model-Based Safety and Assessment, Lecture Notes in Computer Science* (LNCS).

Chapter 4 entitled *Formal Verification of Event-driven Health Applications*, is an article which is published in the *Journal of Software Engineering: Theories and Practices*. It focuses on applying formal verification for an event-driven safety-critical system. This extends the previous work by proposing a technique for verification of temporal properties in a system which is modeled with Stateflow and issues some of external events.

In chapter 5, we propose a technique to facilitate formalizing some LTL properties which can be added to the Simulink block library. This chapter also presents the LTL2SL tool that helps the instrumentation of Simulink models with predefined properties.

Finally, chapters 6 and 7 discuss the techniques proposed in previous chapters, the general results, and possible avenues for future work.

CHAPTER 2 LITERATURE REVIEW

The following chapter puts forth a review of different development environments for safety-critical systems and corresponding development process such as advanced formal verification techniques and tools. We first explain the basic concepts and verification techniques in a categorized format, including modeling concepts and different model checking approaches. The remainder of the chapter will consist of the following: Introducing Simulink tool suit and its block library that the thesis is based on it, followed by describing two different formal verification tools and approaches.

2.1 Modeling Concepts

In order to apply formal methods into development process of safety critical applications, there is a need to construct a formal model of such systems. In addition, development tools in safety critical domain offer a design environment for modeling the system. Following this section modeling concepts are briefly described.

2.1.1 Software Modeling

A model is generally identified as the abstract demonstration of a system. Software modeling is one of the ways to express the software design. To address this, some sort of abstract languages or images are typically used. In addition, the whole software design comprising all software methods, its interactions with other software and interfaces needs to be dealt in software modeling [34]. As such, the system can be modeled by developers with the help of a modeling language which can be textual or graphical [39]. As an illustration, UML which is an object modeling language and can be used to express the software design for an object-oriented language [35].

In addition, System Modeling Language (SysML) [7] is another multi-use modeling language for engineering applications. It is actually a domain specific language which is firstly developed by an open-source specification project for defining the specifications, applying the analysis, design and verification of wide collection of systems.

Since simulation is the execution of a model, by using models at the core of the development process and performing simulation, developers have an intuition into the dynamics and algorithmic characteristics of the system [56].

2.1.2 Model-based Design

Model-based design can be used to facilitate the addressing of difficulties as well as complexities in a control system design. It actually provides an executable specification that indicates the model acts as a functional part of the design process rather than being just a document [54]. It empowers developers to use a single model of their entire system by providing a design environment. In addition, the constructed model can be visualized, analyzed, tested, validated and eventually deployed through the design environment [46]. Additionally, model-based design produces a structure for the software reuse. In particular, it enables that designs being efficiently and dependably upgraded in a more simplistic and cost effective way.

Model-based design as well as automated code generation, are being used gradually at National Aeronautics and Space Administration (NASA) [28]. That is because this kind of design offers several benefits such as higher productivity, increased portability, and elimination of errors that might be caused by manual coding. In addition, MathWorks Simulink® [6] and Simulink Coder™[4] which formerly known as Real-Time Workshop are currently being used by NASA for some part of their modeling and code development [29].

To put it differently, this type of design emphases on using executable system models as the basis for all phases such as specification, design, implementation, test, as well as the verification behboodian2006model. Some parts or all the system specification and requirements in the paper format can be replaced by the executable specification in Model-based design as the main deliverables among design stages. It also contains of an executable model of the application logic that can be simulated. The model elaboration is the next step in the Model-Based Design. In fact, it comprises of some actions to transform the executable specification into a more design based form [60].

The executable model can be built by Simulink® which is a tool that provides an environment for Model-based Design as well as the simulation for dynamic and embedded systems. Moreover, an interactive graphical environment as well as a customizable set of block libraries are provided by Simulink that let the developer to design, simulate, implement, and test a wide range of systems. The Figure 2.1 illustrates some application that Model-based design is used for their development [46].

2.2 Verification and Validation Techniques

Due to advancements in embedded computing technologies, people are incredibly dependent on that technology in home electronic appliances, phones, cars, and more. As a result, those complex software and hardware systems are relied on more than ever in everyday life, and

Aerospace	Automotive	Industrial Automation	Goods and Equipment
DO 178B Certification	Engine emission control	Robotics and automation	Printers
Satellites	Anti-lock brakes	Motor and machine control	Copying machines
Missiles	Climate control, windows	Theme park rides	Mass storage
Autopilots	Automatic transmissions	Upgrades to unsupported legacy systems	Home appliances
	Wire-to-wheel controls		Refrigerators
			Washing machines

Figure 2.1 Model-based Design used in development of real applications

their reliability became gradually essential. Therefore, as demand rises there is a rise in the complexity of the technologies as well as size, and a decrease in development time it has become more difficult to produce said applications with current standards of production. This creates concern over the embedded software's quality. In order to resolve this problem, various verification techniques which can be used for the verification of software, have been introduced. Following this section, verification and validation concepts as well as different verification techniques are briefly described.

2.2.1 Software Verification and Validation

Software producers typically spend large amounts of their total development time and resources for testing the software. In addition, the severity of the flaws also increases the required time for detect the bug. As a result, the cost of the entire development budget for software product increases correspondingly. For this reason, a sophisticated discipline of software engineering is required in order to make sure that all requirements are satisfied by the developed software. In accordance with the IEEE Standard Glossary of Software Engineering Terminology [41]:

- *Verification*: When a system or the component under development is evaluated, to figure out whether the artifacts of the given phase fulfils the requirements imposed at the beginning of that phase, it is called verification process.
- *Validation*: When a system or the component under development is evaluated during or at the end of the development phase, to figure out whether it fulfils specified requirements, it is called validation process.

In other words, software verification is about to guaranty that the software has been developed according to the provided design specifications and requirements. Moreover, as defined by DO-178B standard, verification is performed by applying the reviews, analyses or test [43].

Two fundamental approaches to software verification are: *1) Dynamic verification, 2) Static verification*. Dynamic verification is commonly known as the Software testing, and Static verification applies to a process that formally analyses for proving the correctness of a program to meet the requirements.

2.2.2 Model Checking

Model checking has been developed in early 80's by Clarke and Emerson [25] and also individually has been introduced by Queille and Sifakis [63]. The elementary idea is an automated method to determine if a given specification holds, by exhaustively exploring the reachable states of a program. In other words, it is an automated verification technique that checks whether the given model of a system meets specified properties of the system. If the specified property does not hold in each state, a counterexample as well as an execution trace are produced by the model checker, which leads to a state where the property is violated.

In order to overcome the challenge of directly examining the large state space of software programs, model checking is often combined with abstraction techniques. The *Kripke* structure is the formalism, which is used to represent the system models as a state-transition graph in model checking. A Kripke structure M is a four tuple $M = (S, S_0, T, L)$:

- S is a set of states.
- $S_0 \subseteq S$ is an initial state set.
- $T \subseteq S \times S$ is a transition relation over S, such that for every $s \in S$ there is $s' \in S$ such that $(s, s') \in T$.
- $L : S \to 2^{AP}$ is a labeling function that maps each state to a set of atomic propositions that hold in this state.

The property P is often specified as a temporal logic formula and LTL and Computation Tree Logic (CTL) are two temporal logics which are mostly used. LTL formulas are employed to express the properties pertaining to all paths in the model, while CTL formulas can be used to discriminate among paths.

Symbolic Model Checking

Symbolic model checking is a variation of the traditional format in that it includes the new representation for transition relations [26], and with it the user is able to verify extremely large reactive systems. In symbolic model checking, boolean functions implicitly

represent sets of states. In addition, manipulating Boolean formulas can be done efficiently with Reduced Ordered Binary Decision Diagrams (ROBDD) [20] (or shortly Binary Decision Diagram (BDD)), which is a compact, canonical graph representation of Boolean functions.

This is able to be accomplished due to the number of nodes in the Ordered Binary Decision Diagrams (OBDD) that have to be built, and no longer having to rely on the number of states or size of the transition relation. This has allowed for the possibility to verify reactive systems with realistic complexity.

Bounded Model Checking

Bounded Model Checking (BMC) was first proposed in 1999 by Biere *et al.* [16], and got its name because it involves states that can only be reached during a bounded number of steps. At its core, the BMC is about looking for a counterexample in executions that has a length bounded through the integer k. In the event that a bug is not found then the steps are repeated, but k is increased by one, until the bug is discovered, the issue becomes intractable, or a pre-known upper bound is reached (known as *Completeness Threshold*). By reducing the BMC problem to a propositional satisfiability problem, it can be solved by Satisfiability (SAT) methods instead of Binary Decision Diagram BDDs. Unlike BDD based methods, SAT do not hampered by the space explosion problem. Moreover, they are able to support propositional satisfiability problems that have hundreds of thousands of variables or more. Through experiments, the SAT has shown that it is able to solve many issues that BDD-based techniques cannot. However, BMC has the disadvantage that it cannot prove the absence of errors in many realistic cases [17]. When applying design verification in BMC, the design first unwound for k times, and then conjoined with a property to make a propositional formula, and finally is passed to a SAT solver (Figure 2.2 [30]).

2.2.3 Theorem Proving

Theorem proving is considered as a formal verification technique in which the system and desired properties are represented as formulas in some mathematical logics. Provided by a formal system, this logic defines a set of axioms and inference rules. It is actually the process of discovering a proof for a property, from the axioms of the system. Theorem provers are being used more and more in the mechanical verification of safety-critical properties of hardware and software designs.

According to Clarke et al. in [27], theorem provers for the most part can be divided in a spectrum from *highly automated* to *interactive systems*. The former referring to general-

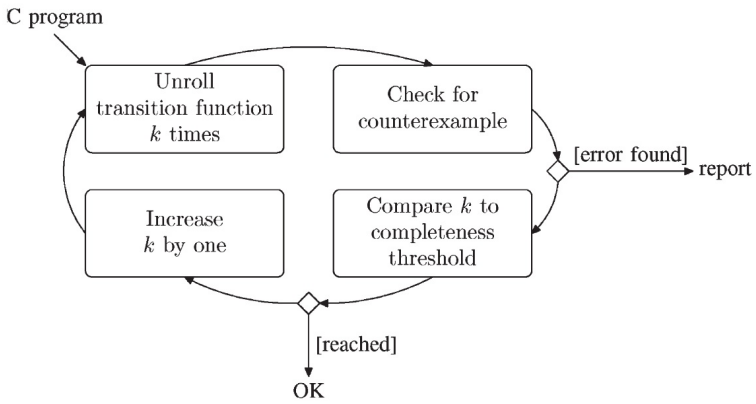


Figure 2.2 High-level overview of BMC

purpose programs and the later referring to special purpose capabilities. The automated systems are beneficial when used for general search procedures and have had a substantial amount of success in solving the different combinational problems. Moreover, the interactive systems are more useful with regards to the systematic formal development of mathematics and mechanizing formal methods.

Since theorem proving does not have the state space explosion problem, it can be used for systems with *infinite* state spaces. This feature makes it different from model checking that suffers from state space explosion problem. There is also a reliance on *structural induction* techniques to prove over infinite domains. However, this technique is flawed in the sense that a human is required for the interactive theorem provers, making the process time consuming and sometimes vulnerable to errors.

2.3 Simulink and Stateflow

By definition, Simulink is a platform for model-based Design and multi-domain simulation of dynamic systems. Stateflow, on the other hand, is a model-based development environment that is widespread and is used in several industries, such as medical, aerospace and automotive. Particularly, Stateflow diagram facilitates the graphical representation of parallel and hierarchical states together with transitions between them and inherits all code and simulation generation capabilities from Matlab toolset. Following this section, Simulink and Stateflow semantics are briefly described.

2.3.1 Simulink

Simulink helps in the design and simulation of wide range of systems by providing an interactive environment along with collections of customizable blocks. It includes extensive library and toolboxes of functions commonly employed in modeling a system.

2.3.2 Simulink model

A physical model can be represented graphically as block diagrams in Simulink environment. This is possible through the use of different blocks hosted in the standard library which is provided by Simulink tool suite. In other words, a Simulink model contains different blocks that are connected through lines as well as some special blocks for communicating to external environment. In addision, Simulink models are stored as '*.mdl*' files, which contains textual description of the model (properties of blocks and their interconnections along with information required for simulation and graphical display of model) [51],[64]. Sample Simulink model is denoted in Figure 2.3.

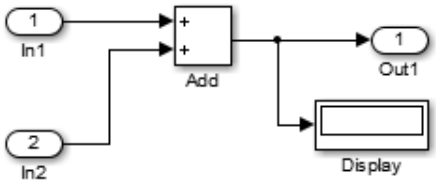


Figure 2.3 Sample Simulink Model

2.3.3 Simulink Library

Simulink comes with a standard block library whose blocks are placed in different categories. To build models in Simulink, blocks are the main elements that are used, and they are hosted in the library. A Simulink block has sets of *input* and *output* ports. A block with N input and M output ports defines a function which describes each of the signals at the output ports as a (possibly time-dependent) expression of the signals at the input ports. Formally, a block is a tuple (P_i, P_o, f), where P_i is the set of input ports, P_o is the set of output ports and $f : \mathbb{R}^{\mathbb{N}} \rightarrow \mathbb{R}^{\mathbb{M}}$ is a function which defines the behavior of the block [52]. In the following, some blocks from the standard Simulink library [6] are briefly described. These blocks are also used in our contribution while specifying properties.

2.3.4 Constant, Inport and Outport blocks

The *Constant* block produces a constant value with the type real or complex, and it can be found in *Sources* category of Simulink library. The constant value as well as the output data type for this block can be defined when designing the model.

Inport block connects a subsystem into an external input by creating an input port. Likewise to *Constant* block, this block can also be found in *Sources* category of Simulink library.

Outport block represents output from a subsystem. In other words, it connects a subsystem to a destination outside of the subsystem by creating an output port. This block and can be found in *Sinks* category of Simulink library.

Simulink® [6] software assigns port numbers for both *Inport* and *Outport* blocks automatically within a top-level system or subsystem sequentially, starting with 1. Figure 2.4, illustrates how above mentioned blocks are represented in Simulink.

2.3.5 Sum block

The *Sum* block applies addition or subtraction on its given inputs, and it is hosted in the *Math Operations* category of Simulink library. It has no state and the sample time for this block is also inherited from driving blocks. This block has two different icon shapes: 1) Round, 2) Rectangular. The Sum block in illustrated in Figure 2.5b has two inputs and one output ports. In this case, the output of the block at the time step t, equals to the addition of both input values at the same time.

2.3.6 Unit Delay block

The *Unit Delay* block holds and delays its given discrete sample time input by the sample period specified as parameter. In other words, *unit-delay* block gives the opportunity to change the sample time of the signal. In addition, he output of this block for the first sampling period is specified using the *initial conditions* parameter.

(a) Constant (b) Inport (c) Outport

Figure 2.4 Source and Sinks blocks

(a) Round (b) Rectangular

Figure 2.5 Sum block

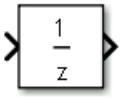


Figure 2.6 Unit Delay

2.3.7 Relational Operator block

The *Relational Operator* block performs a relational operation on its two inputs and produces output. Given operators can be equal, not-equal, smaller than, smaller or equal, greater than, and greater than or equal. Different icon shapes for this block are illustrated in Figure 2.7.

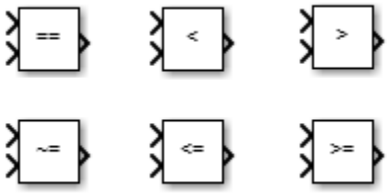


Figure 2.7 Relational Operators

2.3.8 Logical Operator block

The *Logical Operator* block is being used to perform the specified logical operation on its given inputs. The supported operations consist of AND, OR, NAND, NOR, XOR,NXOR and NOT. Table 2.1, represents the supported operations along with their descriptions.

In this block, the number of input ports is specified with the number of input ports parameter. The output type is specified with the Output data type parameter. An output value is 1 if *True* and 0 if *False*.

Figure 2.8, denotes the different icon shapes for this block.

Table 2.1 The supported operations

Operation Item	Description
AND	TRUE if all inputs are TRUE
OR	TRUE if at least one input is TRUE
NAND	TRUE if at least one input is FALSE
NOR	TRUE when no inputs are TRUE
XOR	TRUE if an odd number of inputs are TRUE
NXOR	TRUE if an even number of inputs are TRUE
NOT	TRUE if the input is FALSE

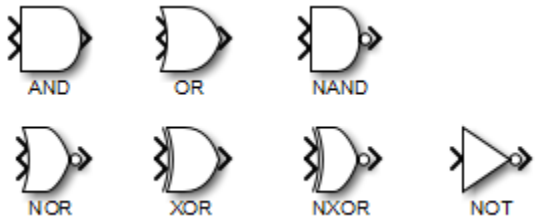


Figure 2.8 Logical Operators

2.3.9 Embedded MATLAB Function block

The Embedded MATLAB Function Block facilitates writing the MATLAB m-code which can be incorporated into a Simulink model. this block is placed in the User Defined Functions Library and can be inserted into a model in the same way as any other Simulink blocks. We use this block whenever there is a need to implement part of the logic of the property by code.

Figure 2.9 MATLAB Function

2.3.10 Subsystem block

A subsystem is a set of blocks that we replace with a single block called a Subsystem block. As the model increases in size and complexity, it can be simplified by grouping blocks into subsystems. In other words, using *Subsystem* blocks can make the system look simpler and

more easier to debug, because these blocks can include other blocks within themselves. Using subsystems has these advantages:

— Assists in decreasing the number of blocks demonstrated in the model window.
— Retains functionally related blocks together.
— Forms a hierarchical block diagram, where a Subsystem block is on one layer and lower layer contains the blocks which made the subsystem.

2.3.11 Function-Call Subsystem block

This block represents a subsystem that can be invoked as a function by another block. In other words, a function-call subsystem is a subsystem that another block can invoke it directly during a simulation. It is similar to a function in a procedural programming language. Invoking a function-call subsystem is equivalent to invoking the output methods of the blocks that the subsystem contains in sorted order. The block that invokes a function-call subsystem is called the function-call initiator. Moreover, the Stateflow, Function-Call Generator, and S-function blocks can all serve as function-call initiators.

2.3.12 Simulink Block Methods

Blocks represent multiple equations which are represented through *Output* and *Update* types known as block methods. Moreover, by running a block diagram these methods are also evaluated.

A simulation loop is used to evaluate block methods in which each cycle through the simulation illustrates a block diagram evaluation at a specific point in time. As such, at the current time step, outputs of each block as well as its states at the previous time step, are calculated by the output method depending on the block inputs. Likewise, discrete state of each block at current and the previous time step are calculated by update method.

A Simulink block has sets of *input* and *output* ports. A block with N input and M output ports defines a function which describes each of the signals at the output ports as a (possibly time-dependent) expression of the signals at the input ports. Formally, a block is a tuple (P_i, P_o, f), where P_i is the set of input ports, P_o is the set of output ports and $f : \mathbb{R}^{\mathbb{N}} \rightarrow \mathbb{R}^{\mathbb{M}}$ is a function which defines the behavior of the block [52, 22].

As an illustration, the *Sum* block in Figure 2.10 has two inputs and one output ports. In particular, the output of this block at time step t, equals to the addition of values of both input ports at time t, and the block function is shown as $m_1(t) = n_1(t) + n_2(t)$.

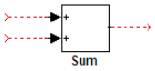
Block	Ports	Function
	$P_i = \{n_1, n_2\}$ $P_o = \{m_1\}$	$m_1(t) = n_1(t) + n_2(t)$

Figure 2.10 Sum block with ports and functions at time $_t$

2.3.13 Simulink Semantics

Simulink has a plethora of semantics (depending on options that are configured by the user), which are informally and partially documented.

Regarding Simulink timing, it is a known fact that the discrete-time Simulink signals are piecewise-constant, continuous-time signals [72]. Associated timing information can be linked to these signals, referred to as *sample time*. Furthermore, the sample time of a signal shows exactly when the signal is updated in the model. When the sample time equals zero, the block is identified as having continuous sample time. This means that, it executes at every point in time. When sample time has a value greater than zero, the block is identified to have discrete sample time.

In Simulink, a discrete block executes at sample time points, and remains constant in the intervals between these sample time points. In like manner, Simulink block methods such as *Output* and *Update* methods are executed at each sample time.

2.3.14 Simulink Block Priorities

Update priorities to blocks can be assigned explicitly. The output methods of each block in the model are executed depending on the their priorities from higher to lower priority. If there is consistency with block sorting rules the priorities can be honoured. Moreover, if the execution order of the block is set explicitly by setting block priorities within a subsystem, Simulink removes those block priority settings when the subsystem is expanded. Simulink checks the block properties in the following order:

— Sample time (faster rate first)
— Priority (lower priority number first)
— Port number (lower input port number first)

2.3.15 Stateflow

Simulink is used to model the continuous dynamics and Stateflow is used to specify the discrete control logic and the modal behavior of the system [71]. The Stateflow modeling language is based on hierarchical state machines with discrete transitions between states. It employs a variant of the finite state notation of machine as established by Harel [38] and offers the elements of language needed to describe complex logic in a readable, natural and understandable form. Given that it is strongly integrated with Simulink and MATLAB, it can offer an environment efficient enough for designing embedded systems that contain supervisory and control.

A state is referred to as *superstate* if there are other states in it and a *substate* when it is held in another state. When a state comprises of two or more substates, it has *decomposition* that can be either parallel (AND) or exclusive (OR) decomposition. All substates at a given level in the hierarchy of the Stateflow must have the same decomposition. In parallel (AND) decomposition, states can be active simultaneously and the activity of each parallel state is independent of all other states.

Defined *Events* can be used to trigger actions in parallel states of a Stateflow chart. One way of triggering an action and/or transition is through broadcasting of an event. The execution of *Actions* can be either as part of a transition from one state to another or based on the activity status of a state which can be ***du****ring* ***ex****it*, ***en****try* and ***on*** *event* actions.

Temporal Operators in Stateflow

These operators are used in the Stateflow and control the execution of a chart in terms of time. In state actions and transitions, you can use two types of temporal logic: event-based and absolute-time. Event-based temporal logic keeps track of recurring events, and absolute-time temporal logic defines time periods based on the simulation time of your chart.

For event-based temporal logic, the following operators *after*, *before*, *every* and *temporalCount* can be used in the Stateflow. For instance, in the porovided model for our case study, the operator *after* is used in *Timers* and *Cylinders* states.

2.4 Verification Tools

This section presents an overview of the modeling and verification tools from different producers. Some of the tools have integrated environment for modeling, simulation and verification. Different tools used different techniques for verification. However, compliance with the stan-

dard for safety-critical application is also considered.

2.4.1 SCADE Design Verifier

Safety-Critical Application Development Environment (SCADE) [3] is a model-based development as well as automatic code generation environment which is based on the synchronous language Lustre. SCADE System has been developed specifically for use on safety critical embedded applications with high dependability requirements. Moreover, different versions are now marketed by Esterel Technologies. As an example, SCADE Suite® is also a model based development environment, which is dedicated to avionics industry. SCADE Design Verifier™is a powerful formal proof engine within the SCADE tool. It uses formal methods and its *formal analysis* portion is based on the Prover Plug-In® [2] formal analysis engine. SCADE Design Verifier enables us to prove that a design is safe with respect to its requirement. In fact, it can prove that something 'bad' will never happen.

SCADE also offers an interactive graphical environment that enables the users to assemble system specifications by dragging and dropping blocks onto a pallet and connecting outputs of one block to inputs of another. By employing the integrated Safe State Machine (SSM) add-on, control logic for representing system states and state transitions can be modelled. SCADE can generate C source code by using the KCG™code generator which has been qualified as a *Level A* software development tool in accordance with RTCA/DO-178B [43]. In addition, SCADE Suite also includes a gateway that can import Simulink models.

2.4.2 Simulink Design Verifier

Simulink Design Verifier® [5] is a tool set of Matlab which uses *formal methods* to identify hard to find *design errors* in the models without requiring extensive tests or simulation runs. It uses the Prover Plug-In® [2] *formal analysis* engine, in order to prove the properties. It is known from the documentation of the SLDV, that the Prover Plug-In is based on Stålmarck's proof procedure which was patented in 1992 [68]. In addition, performing bounded and unbounded model checking, sequential and combinational equivalence checking as well as the test generation are some features provided by this tool. Moreover, modeling of sequential systems employing imperative and declarative formalisms is also supported by prover engines. It also supports a wide range of data types including integers, reals, arrays and booleans. Design errors that can be detected by this tool are as follows:

— *Dead logic*: Discovers the certain designed functionality that can never be activated.
— *Integer overflow*: If certain valid input data causes non-deterministic behavior.

— *Division by zero*: Identifies whether this situation happens early in design time.
— *Violations of design properties*: Verifies the design against requirements.
— *Assertions*: Detects faulty behavior by using the assertion block.

Blocks in the model are highlighted by the Simulink Design Verifier, containing the above mentioned errors. For every block with an error, it calculates signal-range boundaries and generates a *test vector* that reproduces the error in simulation.

Simulink Design Verifier enables us to accomplish model analysis within the Simulink environment, in order to verify the system design and validate the requirements earlier. We can use Simulink, MATLAB functions, and Stateflow to express formal requirements. The provided block library, includes test objectives, proof objectives, assertion, and a set of temporal operator blocks for modeling of systems with temporal characteristics. Furthermore, model coverage objectives can be chosen as: Condition, Decision, and Modified Condition/Decision Coverage (MC/DC).

Also, the MathWorks *DO Qualification Kit* product, helps applying the tool qualification for RTCA/DO-178B [43] and related standards. The model Advisor also checks modelling standards for DO-178B and detect, troubleshoot modelling and code-generation issues.

2.5 Formal Verification of Simulink Models

Many projects currently use MathWorks Simulink and Simulink Coder [9] which formerly known as Real-Time Workshop for at least some of their modeling and code development [29]. This kind of Design focuses on using executable system models as the foundation for the specification, design, implementation, test, and verification [13]. The executable specification in Model-based design, replaces parts or all of the paper format of the system specification and requirements as the main deliverable between design stages. It consists of an executable model of the application algorithm that can be simulated. The next step of Model-Based Design is known as model elaboration which consists of transforming the executable specification into a more design based form [60].

2.5.1 Approach based on other Model Checking Tools

Simulink is a platform for model-based Design of embedded systems from Mathworks. Simulink Design Verifier is a toolset that uses formal analysis for property proving. Although Simulink design Verifier provides some temporal operator blocks in its library, but specifying LTL properties is not potentially straightforward. For this reason, proving linear temporal properties

in a Simulink model is done by transforming the model into the input language of another model checkers.

For the verification of properties, there are works describing the translation of Simulink models into the various model checking languages, including NuSMV, Lustre, SAL and Promela/SPIN [55, 53, 72, 59, 50].

The primary motivation in [53] presenting a translator algorithm along with a tool that can automatically translate a subset of Simulink model into NuSMV model checker as an input language. Meenkashi et al. believe that using the proposed tool shortens the process of formal verification of safety avionics components with less error.

Christian Heinzemann et al. in [40], proposed a model-2-text transformation that is used to transform instances of the given Simulink EMF-Model. Similarly, Pajic et al. In [58], presented a matlab plug-in tool known as UPP2SF that can be used for the translation of models from UPPAAL to Simulink/Stateflow. The proposed tool also enables UPPAAL models to be simulated and tested in Simulink/Stateflow.

With attention to linear temporal properties, efforts have been exerted recently in describing LTL verification of Simulink models. As an illustration, Miller et al. [55] propose using the symbolic model checker NuSMV [23] as the verification tool. Since the simulink model can not be used directly as an input for NuSMV model checker, as a result it should be first being translated into synchronous dataflow language Lustre [36] and then into NuSMV. Similarly, the primary motivation in [12] was to verify the correctness of Simulink models with respect to a set of specifications given as LTL formulae. The authors applied the explicit model checking technique, after initially formalizing the simulink models based on the set-based reduction concept. it is done to reduce the state space and support for non-determinism of input.

Roy et al. in [65], defined an approach that uses a contract system for partial verification of the given Simulink model. They used the contract annotation language to capture constraints on signals along with the relations between them. Furthermore, annotations were translated into verification conditions. In order to apply the verification using Yices SMT solver. For this reason, annotations finally translated into the constraint language of Yices.

Authors in [67], employed the SPIN model checker for verification of the Simulink model including the Stateflow.

Leitner in [49], performed and evaluation between Simulink Design Verifier and the SPIN model checker by using a NVRAM case study. The safety properties are specified in Simulink Design Verifier, but for the LTL properties the author used the SPIN model checker.

2.5.2 Approach based on Simulink Design Verifier

There has been recent developments to use formal methods in order to design critical systems. A prime example of this is with Jian *et al.* [42] who developed a Virtual Heart Model (VHM) that works in real-time in order to model the electro-physiological operation of proper functioning and malfunctioning. Through this procedure the team used Simulink Design Verifier to design a timed-automaton model to define the timing properties of a heart.

Similarly, Simulink and Stateflow have been employed in [32] in order to model the tracking function of trains in an automatic train protection system. It was implemented using the provided requirements specification document which has the safety and functional properties presented in natural language. The authors of [32] claimed to have had a positive experience when they used Simulink Design Verifier in the train transportation field and they also used it for verification as well as validation.

In [57], another instance of using Simulink with a medical device has been demonstrated in which an iterative approach is used in system verification based on the software architectural model. In this instance, Simulink/Stateflow is used for describing the behavior of the model at a component level. Moreover, authors employed the Simulink Design Verifier for establishing component-level properties by proving the system level properties.

In [11], the Fuel Level Display System of the Scania is modeled in Simulink and safety requirements are verified using Simulink Design Verifier.

Bergquist et al. in [15], modeled an electronic climate control module of a car in Simulink. The Simulink model then instrumented with some assertion based properties, and finally Simulink Design Verifier is used as the verification tool.

2.6 Conclusion

Various tools and techniques may be used for modeling, simulation and verification of the safety-critical applications. Each technique might have its own limitations, consequently the tools that use those techniques may have the same limitations.

In terms of verification technique and comparison to model checking, theorem proving can be applied to systems with *infinite* state spaces because it doesn't have the state space explosion problem like the one that other model checking technique has.

In terms of theorem proving in two well suited tools, it should be noticed that, both Simulink Design Verifier® [5] and SCADE Suite Design Verifier™[3], use the Prover Plug-In® [2] formal analysis engine, in order to prove the properties. This Plug-In is developed and

maintained by Prover Technology and can be used to perform bounded and unbounded model checking, combinational and sequential equivalence checking, and test generation.

Simulink Design Verifier uses Simulink and Stateflow for model-based design and create the models, in fact, they both use the same development environment. SCADE models can be described within the SCADE model editor. Moreover, the SCADE software model can be automatically translated from Simulink through the SCADE Simulink Gateway or UML Rhapsody through the SCADE UML Gateway. In terms of limitations of Simulink Design Verifier, below items illustrate some of major limitations:

— Models containing algebraic loops are not supported
— Supports only fixed-step solvers and discrete time system (not variable-step solvers that are used for continuous time systems)
— Supports only real signals (not Complex signals which consist of a real part and an imaginary part)
— Recursion in Stateflow is not supported due to not supporting the recursive functions by Simulink Design Verifier (SLDV). Although, the Stateflow software allows developer to create *cyclic behavior* (a sequence of steps is repeated indefinitely), but if the model has a chart with cyclic behavior, SLDV cannot analyse it.

As a benefit, Simulink Design Verifier provides a mechanism that checks whether the model is compatible for analysis. Otherwise, it alerts the developer to any incompatibilities that it identifies in the model. It also performs design error detection on the given Simulink model. As an illustration, integer overflow, division by zero, dead logic, and assertion violations are such design errors that are supported by Simulink Design Verifier.

The existing verification blocks in the block library provided by Simulink Design Verifier, are suitable for modeling of safety requirements. For this reason, some efforts have been exerted in transforming Simulink models into the input language of other model checkers to be able to specify LTL properties. This thesis aims to enrich the development process provided by Simulink tool suite. To address this, it proposes formal verification of Simulink models based on extending the Simulink library with some customizable blocks. In other words, the idea is to translate the LTL propertiy into an invariant by using existing blocks. In addition, the proposed blocks have the support for some LTL properties and will facilitate the process of instrumenting models with properties. As a benefit of this approach, we can mention that Simulink Design Verifier and Simulink are integrated in the same environment. The specified LTL properties in the Simulink model can be analysed by Simulink Design Verifier, so the entire process has less step due to no need for transforming the model.

The next three chapters are represented by three articles that deal with formalization and verification of some safety and (timed) linear properties using the Simulink tool suite.

CHAPTER 3 ARTICLE 1
Applying Formal Methods into Safety-Critical Health Applications

Mohammad-Reza Gholami, Hanifa Boucheneb
Model-Based Safety and Assessment, Lecture Notes in Computer Science (LNCS)

Abstract– Software performs a critical role in almost every aspect of our daily life specially in the embedded systems of medical equipments. A key goal of software engineering is to make it possible for developers to construct systems that operate reliably regardless of their complexity. In this paper, by employing model-based design for large and safety-related applications and applying formal verification techniques, we define specific properties to ensure that a software system satisfies its correctness criteria. We use the formal approach to study and verify the properties of a medical device known as Endotracheal intubation. We present how the system is modeled in the Simulink and Stateflow and present a functionality formalization. In order to formally prove some critical properties, we employ Simulink Design Verifier toolset.

Keywords– Formal Methods, Formal Verification, Design Verification, Model-Based Design, Safety-Critical.

3.1 Introduction

The increasing complexity of embedded systems (e.g. avionics, health and automotive systems) conveys the producers of safety-critical applications to use more systematic processes for development. Traditional design processes are not fast enough in discovering the errors in requirements; hence, the whole process would be longer and more expensive.

A key goal of software engineering is to construct systems that operate reliably regardless of their complexity. A promising approach to achieve this goal is to use formal methods, which are mathematically based languages, tools and techniques for specifying and verifying such systems. Formal methods can significantly increase our understanding of a system by disclosing inconsistencies, ambiguities, and incompleteness in the early design phase so that they can be eliminated in order to ensure the appropriate behavior of the system.

Software vendors typically spend large amounts of time of their total development time and resources on software testing. The severity of the defects also increases the detection time and the cost of total development budget for a software product. So there is a need for a

sophisticated discipline of software engineering to ensure that all the expected requirements were satisfied by means of the specified software. In other words, software verification refers to the process that can determine whether the artifacts of one software-development phase fulfil the specified requirements produced during the previous phase.

In a safety critical system, even a single error in the source code and an associated malfunction of the system can cause high costs and could even endanger the health of people. In critical real-time embedded systems, verification and validation activities are becoming huge and quite costly when the complexity and size of the systems grows.

Formal verification can be performed by employing Model-Based Design [33] in order to specify formal model of the system. Model-Based design facilitates the addressing of difficulties and complexities existing in control system design by providing an executable specification which implies that the model exists as more than just a document, but as a functional part of the design process [54]. It also provides a single design environment that enables developers to use a single model of their entire system for data analysis, model visualization, testing and validation, and ultimately product deployment, with or without automatic code generation [46]. Furthermore, model-based design creates a structure for software reuse that permits established designs to be effectively and reliably upgraded in a more simplistic and cost effective manner.

In this paper we are using the formal approach to study and verify the properties of an Endotracheal intubation which is a specific type of tracheal tube that is inserted through the patient's mouth to maintain an open airway [1]. This medical device is illustrated in Figure 3.1. Using Simulink/Stateflow, we come up with a model with parallel components, where event passing and synchronization is efficiently provided.

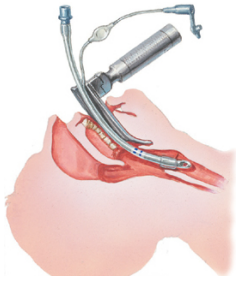


Figure 3.1 Endotracheal intubation

The rest of the paper is organized as follows: Section 3.2 presents some background and previous work related to this research topic. Employed tools are described in Section 3.3. We present our case study in Section 3.4, its properties and implementation of the model. The

outcomes of our implementation are analysed in Section 3.5. Finally, Section 3.6 concludes our paper.

3.2 Background and Related Work

Formal methods enhance verification process by using formal notations and concepts in writing requirements and specifications. In formal methods, mathematical and logical techniques are used to express, investigate, and analyze the specification, design, documentation, and behavior of both hardware and software. The word *formal* in formal methods derives from *formal logic* and means "to do with form" [66]. In formal logic, dependence on human intuition and judgment is avoided in evaluating the arguments. In order to constitute an acceptable statement or a valid proof, formal logic employs a restricted language with very precise rules for writing assumptions, theorems, and proofs. In formal methods for computer science, languages are enriched with some of the ideas from programming languages and are called *specification languages*, but their underlying interpretation is usually based on a standard logic.

Formal verification in the field of software means the automated proof of specified properties on the code without executing the program. Also, it ensures that a design conforms to some precisely expressed notion of functional correctness [18]. The main benefits of formal verification in comparison to testing (dynamic verification), are its soundness and exhaustiveness. Specifications in formal methods are *well-formed* mathematical statements which are used to specify a property that needs to be verified in the system [10].

Many projects currently use MathWorks Simulink and Simulink Coder [9] which formerly known as Real-Time Workshop for at least some of their modeling and code development [29]. This kind of Design focuses on using executable system models as the foundation for the specification, design, implementation, test, and verification [13]. The executable model replaces parts or all of the paper format of the system specification and requirements as the main deliverable between design stages. It consists of an executable model of the application algorithm that can be simulated. The next step of Model-Based Design is known as model elaboration which consists of transforming the executable specification into a more design based form [60]. In Section 3.3, we briefly explain how to build an executable model by using Simulink and Stateflow.

Recently, some efforts have been made in order to employ formal methods in designing critical systems. In particular, Jiang *et al.* [42] developed a real-time Virtual Heart Model (VHM) for modeling the electro-physiological operation of proper functioning and malfunctioning.

They introduced a timed-automaton model to define the timing properties of the heart and used Simulink Design Verifier as the main tool for designing their model.

Simulink/Stateflow has also been used in [32] to model a train tracking function for an automatic train protection system. The model was implemented based on the requirements specification document in which safety and functional properties were originally written in natural language. The authors of [32] used Simulink Design Verifier for verification and validation. They also had a positive experience when they used this tool for the safety-critical function in the railway transportation domain.

Another case example for a medical device has been presented in [57] where an iterative approach is applied for system verification based on software architectural model. They employed Simulink/Stateflow for describing the component level behavior of the model and used Simulink Design Verifier for proving the system level properties to establish component-level properties.

In our work, we also use Simulink/Stateflow to model and simulate the system. For the formal analysis, we use Simulink Design Verifier, which intensively employs the BMC and K-Induction features of the PROVER[2] engine to establish the satisfiability of the proof objectives. We also employ this tool to verify *Integer overflow*, *Division by zero*, *Assertions and Violations of design properties* as a part of our work.

3.3 Simulink and Stateflow

The executable model can be built by Simulink which is an environment for multi-domain simulation and Model-based Design for dynamic and embedded systems. Mode logic in Simulink models is described in terms of hierarchical state machines specified in a variant of Statecharts called Stateflow [9].

Stateflow is a widespread model-based development environment in Matlab/Simulink toolset, which is used in several industries, such as aerospace, medical, and automotive. It uses a variant of the finite state machine notation established by Harel [38] and provides the language elements required to describe complex logic in a natural, readable, and understandable form. Since it is tightly integrated with MATLAB and Simulink, it can provide an efficient environment for designing embedded systems that contain control and supervisory. In particular, Stateflow diagram enables the graphical representation of hierarchical and parallel states as well as transitions between them and inherits all simulation and code generation capabilities from Matlab toolset.

A state is called as *superstate* when it contains other states and a state is called *substate*

when it is contained by a superstate. When a state consists of one or more substates, it has *decomposition* that can be either parallel (AND) or exclusive (OR) decomposition. All substates at a particular level in the hierarchy of the stateflow must have the same decomposition.

In parallel (AND) decomposition, states can be active at the same time and the activity of each parallel state is essentially independent of other states.

We can use our defined *Events* to trigger actions in parallel states of a Stateflow chart. Broadcasting of an event can trigger a transition and/or an action. The *Actions* can be executed either as part of a transition from one state to another or based on the activity status of a state which can be entry, during, exit, and on event actions.

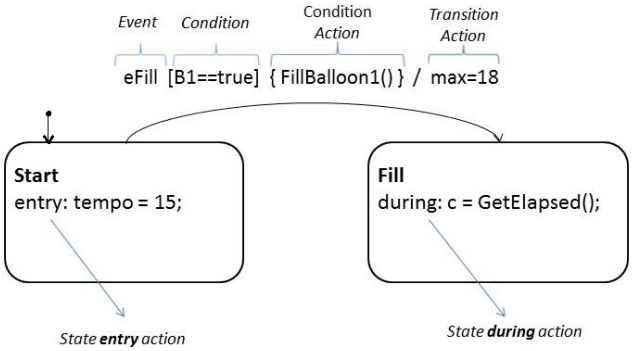


Figure 3.2 Stateflow Semantics

The general form of a transition in Stateflow is presented in Figure 3.2. It shows the behavior of a simple event, condition and transition action specified on a transition from one exclusive (OR) state to another. Initially, state *Start* is active and *Entry* action is executed. When the event *eFill* is received, the chart root detects that there is a valid transition to state *Fill* as a result of the event *eFill*, so it validates the condition and if the result is true, the *Condition Action* immediately gets executed and completed. The state *Start* is marked as inactive and the *Transition Action* is executed and completed when the transition destination *Fill* has been determined to be valid. States can have different actions such as: *entry*, *during*, *exit*, and *on event-name* which are being executed based on the current status of the active state.

3.4 Case Study

This case study aims to show and familiarize how a system works with a framework where time aspects are combined with multi task programming. In this case study, we are modeling and verifying the properties of a Filling system of balloons of an intubation probe. An

intubation probe is placed to ensure continuous passage of air to the lungs and introduce oxygen sensors, aspiration probes to the lung for patient treatment. This system consists of two balloons, two access valves for manual inflation, two pressure sensors, a power distributor, a pump and an air tank. The pump is actuated by a gear motor and a transmission by a cylinder rack. The pumped air is propelled through the power distributor (B and D) to one of the balloons or outside. The probe has several buttons (*Start*, *Stop*, *Duration*, *Pressure*, *StopAlarm*) and LEDs (*L1*, *L2*, *Alarm*). The Alarm LED reports the anomalies. L1 and L2 are witnesses indicating the inflated balloons, and the button *StopAlarm* allows the user to stop the alarm. The system controlled by a Programmable Controller who is responsible for controlling the commands and messages which are sent to or received by other components. We are using Simulink and Stateflow as an integrated tool environment for modeling, and Simulink Design Verifier for verification of some properties.

3.4.1 The model

A model is known as abstract representation of a system. Software model is actually the ways of expressing a software design, and in order to express the software design some kind of abstract language or pictures are usually used. Software modeling need to deal with the entire software design, including interfaces, interactions with other software, and all the software methods [34]. Engineers can model the system using a modeling language which it can be graphical or textual [39].

According to the description of the case study, the first step when modeling the Intubation, is to pinpoint the superstates in the system. One of the most important parts of the design is to find out which superstates should be parallel (AND) and which ones should be exclusive (OR).

In the Intubation statechart there are ten distinguished blocks which are illustrated in Figure 3.3. All of these blocks are working in a parallel execution order. These blocks are represented with ten different superstates in the model. In every moment of running the model, at least one state has to be active in each superstate. The superstates are: *Controller*, *Distributor*, *Cylinder*, *Balloons*, *Alarm*, *Lights* and *Timers*. These superstates are designed to be parallel (AND) because a change in these states is allowed at every time step.

The superstates interact together through sending direct broadcast event and one simplified function to make the model smaller, initializing the variables and status. Direct event broadcasting is used to prevent receiving an error pertaining to recursion in the Stateflow chart. In figures, some functions are removed because of simplicity. To explain and describe the model, this section is divided into three different parts as follows:

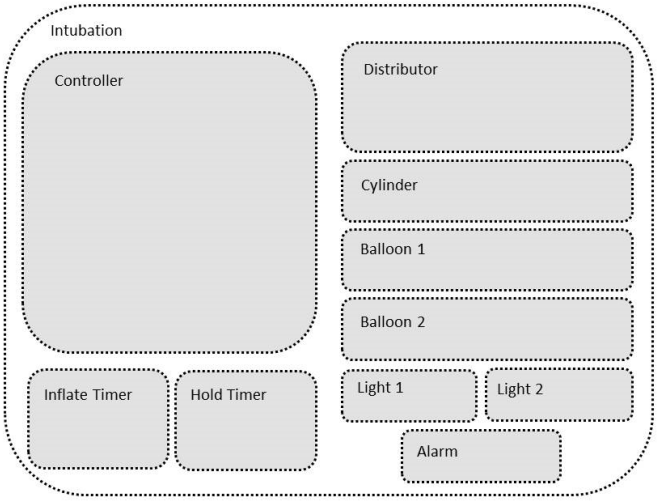


Figure 3.3 Components of the Endotracheal Intubation

- Inputs & Local variables, ranges and default values
- Events
- Components (Parallel (AND) States)

In the following we will give a short description of the components. We include some screenshots taken from Simulink to complement the description of the statecharts.

Variables:
We maintain the current state and values of the processes using local variables. For simplification, we used integer values to represent the corresponding physical step of the components. The input type variable corresponds to the variable whose value is coming from the Simulink model. Conversely, the output type variable is modified and used in the Stateflow, and it is accessible through the Simulink model. Both input and output type variables, their range, and default values are defined and set in declaration of the stateflow, for example, for the cylinder component:

$int\ nCylPos = -1$

For this specific variable, the value can be set to either -1, 0 or 1 which respectively corresponds to the position of the cylinder as: *Back*, *Center* and *Forward*. Similarly, the different values for *nBallonState* correspond to different status of the balloon, such as: *Empty*, *Not-Full* and *Full*. The target pressure of the balloon in which the balloon is considered as full inflated, is stored in *nPressure*. The variable *nDuration*, sets the amount of the time that an inflated balloon should remain full before controller sends a deflation command.

Some input and output variables for the Stateflow diagram are listed in Table 3.1.

Table 3.1 Variables in Intubation Stateflow

Name	Type	Values
nCylPos	Output	-1, 0, 1, 2
nBallonState	Output	-1, 0, 1
bLightState	Output	true, false
bAlarm	Output	true, false
nDuration	Input	10, 20, 30 mins
nPressure	Input	12, 18, 24
bStopAlarm	Input	true, false
bStart	Input	true, false

Events:
Different events were defined to model the communication between different components of the system. According to their usage, they are defined in the Stateflow as *Directed Event Broadcast*. The relationship between events and the corresponding component that receives these events, is listed in the Table 3.2.

Table 3.2 Events

Component	Events
Alarm	eStartAlarm, eStopAlarm
Lights	eLight1On, eLight1Off, eLight2On, eLight2Off
Distributor	eFill, eEmpty
Cylinder	vPlus, vMinus

As an illustration, the event *eStartAlarm* is sent by the controller whenever the inflation or deflation period of a particular balloon exceeds the specified time (15 Seconds). Similarly, *eFill* and *eEmpty* events, are sent to *Distributor* state (along with appropriate values for B and D), for every request for inflating and deflating of a particular balloon.

Stateflow:
This section describes each component that we have modeled as superstates. We explain the nature and the interaction of each one of them.

Controller

The model of the controller was realized from the specification based on the provided Grafcet [61]. In order to model the controller, some local variables representing the steps, transitions

and actions and some local variables representing the channels and reflecting the value of local actions in the system are also defined. The local variables B and D are defined as *boolean*, and being used to set the channels and activate them.

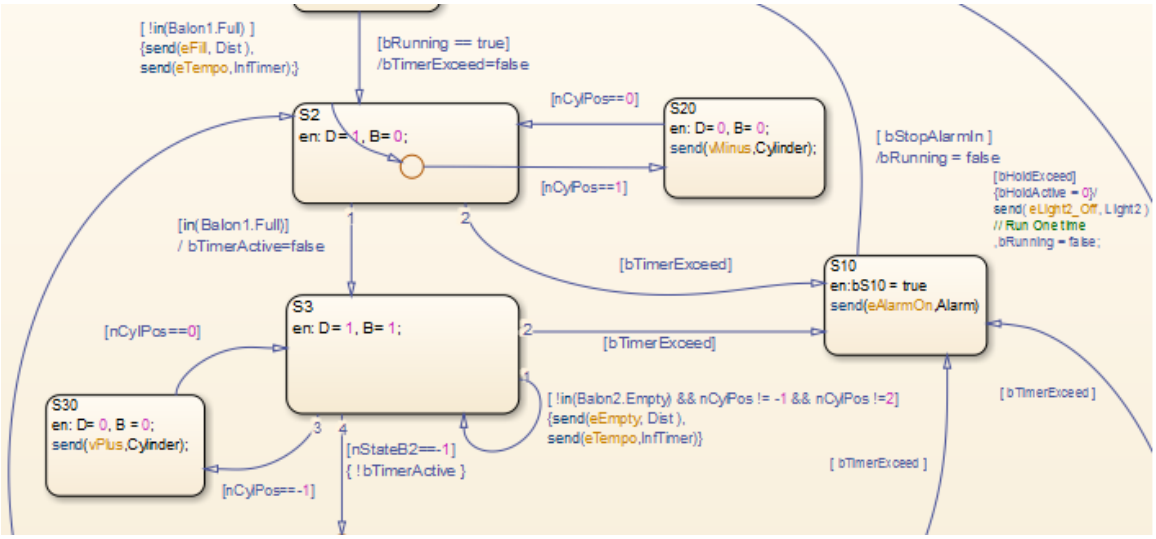


Figure 3.4 Part of Controller State

The entire controller is included in a single superstate. In order to do a specific action, the controller sets the values for the channels and sends the event to the Distributor state. Table 3.3 presents events and channels used by the controller for the specific actions.

Instead of modeling the user's interaction with the system. We use local variables to simulate the choice of target pressure and the targeted cycle time (*nPressure* and *nDuration*). Those values are set at the beginning and are left untouched during the simulation. Our model assumes that at the beginning the cylinder is in the position *Back* and both balloons are considered *Empty*.

One of the major building blocks of our controller which is responsible to fill and empty the balloons is illustrated in Figure 3.4. In this figure, in order to inflate the first balloon, on the entry action of the state *S2* corresponded values for B and D are set then the event *eFill* is sent to the *Distributor*.

Cylinder

The cylinder has three substates: *Back*, *Center* and *Forward*. This state is constantly waiting to receive the events *vPlus* or *vMinus* from the controller to change its position forward or backward. To model the 2 seconds delay for each position transition, we included an in-between location, and used an *after* temporal operator between each position. Once the

delay is exhausted, the cylinder position changes. We use a local variable *nCylPos* (*Back* = -1, *Center* = 0, *Forward* = 1 and *In-between* = 2) to store the current position of the cylinder. Initially, the state *Back* is active. We set this variable to 2 when the cylinder is in transition between two positions. The controller has guards using the transitional value to ensure the cylinder has completed its movement.

Pressure Distributor

This state receives the specified event from the controller in order to launch the selected action for filling or emptying the desired balloon. The selected action is relevant to the current value of the local variables *B* and *D* which are set to *true* or *false* by the controller before sending the event *eFill* or *eEmpty*. In addition, to complete the entire function, it also sends specific events to states *Cylinder* and *Balloon* for their relevant actions. Table 3.3 shows the events and variables used for the specific functions.

Table 3.3 Actions

Distributor Channel	Event	Function
B=0, D=1	vPlus	Inflate ballon 1
B=1, D=0	vPlus	Inflate ballon 2
D=0 or B=0, D=0	vPlus	Move cylinder forward
B=0, D=1	vMinus	Deflate ballon 1
B=1, D=1	vMinus	Deflate ballon 2
D=0 or B=0, D=0	vMinus	Move cylinder backward

Initially, the state *Init* is active and waits to receive a specific event from *Controller* to complete the selected function. The state on the rightmost side of Figure 3.5 has to run iteratively until the destined balloon is filled. Similarly, the emptiness of the balloon is ensured by running the state on the leftmost side of this figure.

Balloons

The state balloon has three substates: *Empty*, *NotFull* and *Full*. In this model, we consider two different superstates corresponding to each balloon. Initially, the state *Empty* is active in both balloons. The transition between substates has a guard; so, the movement is done when one of the events *eFill* or *eEmpty* is received from the state *Distributor*. We use a local variable *nBalloonState* (*Empty* = -1, *NotFull* = 0, *Full* = 1) to store the current status of the balloon.

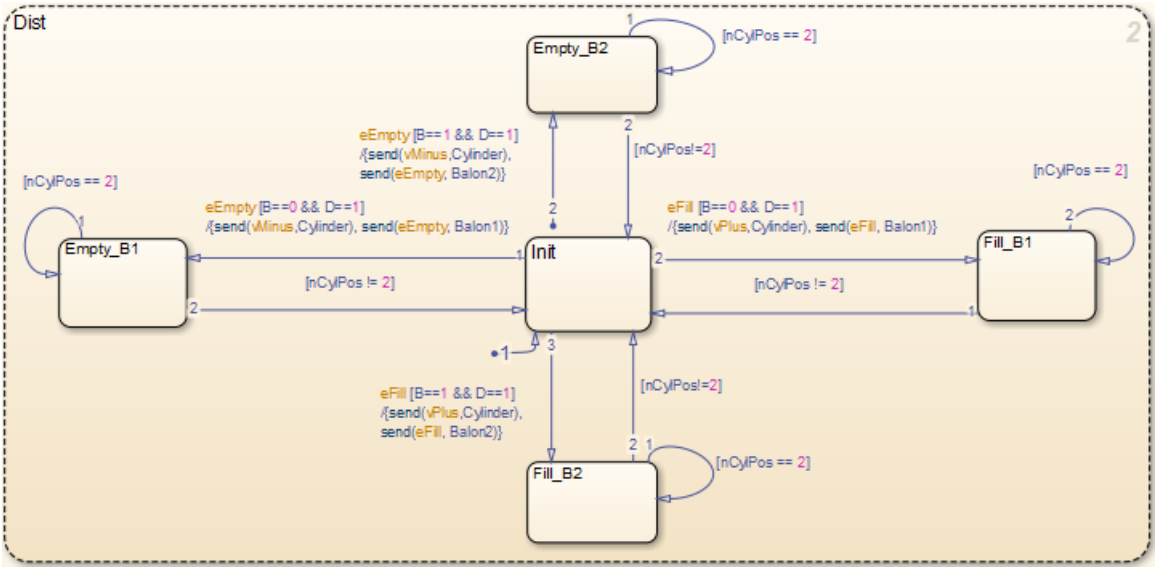


Figure 3.5 Pressure distributor

Timers

In our model, timers are designed as two different components: The *InflateTimer* which is responsible for the inflation time of a balloon, and the *HoldTimer* which is the time that a balloon should maintain the status *Full*.

The state *InflateTimer* contains two substates: State *Off* which is initially active, and the state *On* which is activated by the controller while requesting for a *Fill* function. The *InflateTimer* is illustrated in Figure 3.6:

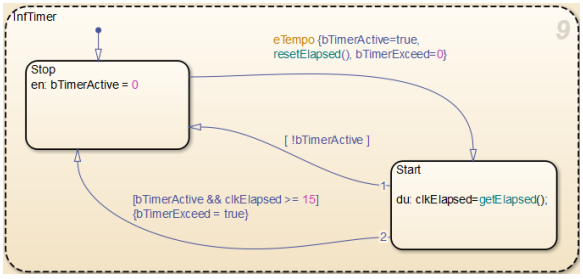


Figure 3.6 Timer State

Alarm

The alert state contains two substates: *Off* and *On*. Initially, the state *Off* is active. When the timer exceeds from the specified threshold or if any anomaly happens, the controller stops

the system operation and sends the event *eAlarm*. Once the event received, the state *On* will be activated and remains in this state until the user stops the alarm.

3.5 Results and Analysis

This section describes our method and results of formal verification using Design Verifier with Simulink and Stateflow. Before verification, we run the simulation for our provided model using predefined input parameters in order to ensure that the model can be executed properly. Figure 3.7 illustrates an execution snapshot of our Simulink implementation for one of the system properties which states that two lights should not be turned on at the same time. Similarly, Figure 3.8 validates the emptiness property of each balloon, meaning that a balloon's light is *off* when the balloon is completely empty.

The values 0 or 1 for the lights shows that the light is *Off* or *On* at corresponding time step. In addition, the values -1, 0, 1 correspond to *Empty, NotFull, and Full* status for balloons. As illustrated in Figure 3.8, at time step 1200, the status for balloon 1 becomes *Empty* (-1), and as a result the corresponding light in Figure 3.7 becomes *Off* (0).

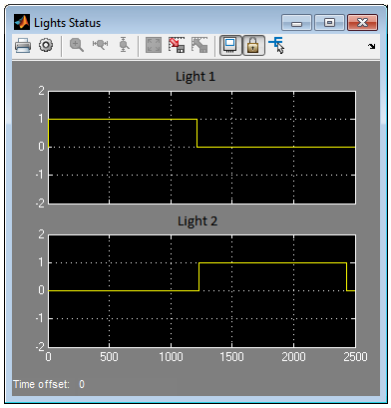


Figure 3.7 Lights status

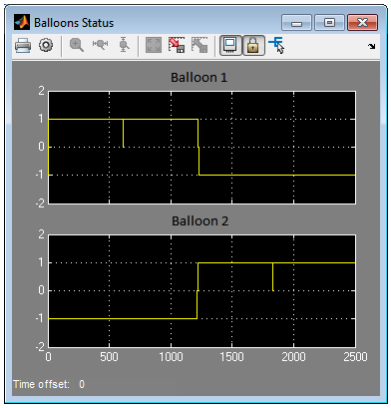


Figure 3.8 Balloons status

3.5.1 Properties

The term property refers to a logical expression of signal values in a model. For example, we can specify that a signal in a model should attain a particular value during execution of the system. The Simulink Design Verifier software can then prove the validity of such properties. This is done by performing a formal analysis of the model to prove or disprove the specified properties. If the software disproves a property, it provides a counterexample that demonstrates a property violation. Our design model consists of the following properties:

1. Balloons should not be inflated simultaneously for more than five time steps.
2. The pressure in each balloon never exceeds a predetermined value.
3. Any anomaly is followed by alarm activation.
4. The lights *L1* and *L2* are never illuminated simultaneously.
5. There must be no anomaly alarm (False alarms).
6. If a light is *ON* then the corresponding balloon is inflated.

The following section details each property and the results obtained by Simulink Design Verifier:

Property I: The goal of this property is to ensure that two balloons are not inflated simultaneously more than the accepted time. Although the controller sends appropriate commands to inflate a balloon and deflate another, but the functionality also depends to the position of the cylinder and current state of each balloons. So, verifying this property ensures this time will not exceed the expected time steps. As illustrated in Figure 3.9, we use a temporal operator *Detector* in formalization of this property. This property is proven *Valid* with values greater than 5, and is proven *Falsified* with values between 1 and 5.

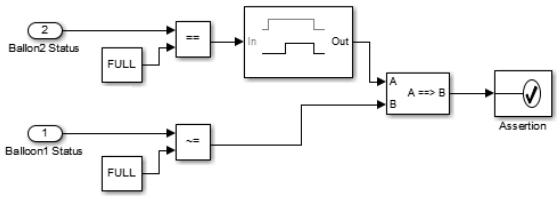


Figure 3.9 Formalization of property I

Property II: The goal of this property is to validate that the model do not permit the pressure within the two balloons to exceed the maximum value given by the user. This property can be defined as the following expression in LTL:

$$\begin{aligned} G\ (&nPressB1 <= nPressure \\ &\&\&\ nPressB2 <= nPressure) \end{aligned} \tag{3.1}$$

We have modeled this property by doing a simple comparison between the balloon pressure and the target pressure using our variables *nPressB1* and *nPressB2* which contains the current pressure within the balloons 1 and 2, and by using *nPressure* which represents the

target pressure given by the user. We validate both pressures by using the same property statement.

Property III: The goal of this property is to make sure our model does not skip any alarms. Therefore all possible anomaly should be followed by an alarm activation. In our model, the alarm is triggered by the event *eAlarmOn*. It automatically lights the Alarm indicator. We have modeled this property by using the Alarm 'On' state and the controller's anomaly state 'S10' which is a location where all anomaly detections are directed.

$$\begin{aligned} G\ ((bRunning == true\ \&\&\ in(Controller.S10)) \\ \Rightarrow\ Alarm.On\ ) \end{aligned} \tag{3.2}$$

This property validates that our model is running and then makes sure that all paths that goes through 'S10' are followed by the 'Alarm.On' state.

Property IV: The goal of this property is to make sure that both balloon's status light are not both lighted at the same moment.

$$G\ ((Light1.On + Light2.On) <= 1) \tag{3.3}$$

Since this property is a safety property, all paths needs to validate it. As we saw in property I, we may have both balloons inflated for a short period of time. Therefore, in order to validate this property, we have to make sure that the controller have control over the lights. The controller always starts by closing a light, and then opens another. This is validated by making the sum of values in states 'Light1.On' and 'Light2.On' which should not exceed 1.

Property V: The goal of this property is to make sure our model does not generate false alarms. To address this property we have designed a statement that validates the opposite condition and applied a *'not'* to it.

$$\begin{aligned} G\ (not\ (bRunning == true \\ \&\&\ bAlarmDetected == false \\ \&\&\ Alarm.On)) \end{aligned} \tag{3.4}$$

This invariant property makes sure that we never end up in a state where we have *'bRunning*

== true' (which means the process is undergoing) and we have the alarm light on without having detected any anomaly. The anomaly detection always sets the variable *'bAlarmDetected'* to *true*. Hence, if *'bAlarmDetected'* is set to *false* we have no anomaly and 'Alarm.Off' is valid.

Property VI: The goal of this property is to ensure that when a light for a balloon is on, the corresponding balloon's pressure has reached the target pressure.

$$\begin{aligned} G\ ((Light1.On \Rightarrow nPressB1 == nPressure) \\ and\ (Light2.On \Rightarrow nPressB2 == nPressure)) \end{aligned} \tag{3.5}$$

We have included both balloons in the same property, where they both need to be *true*. This statement validates that when the state 'Light1.On' is active, we have reached the target pressure in the first balloon. It does the same validation for the second balloon and ensures that both condition are always valid.

3.6 Conclusion

In this paper, we used formal approach and model-based design in order to specify and formally verify the functionalities of a medical device. The system is modeled with parallel components in Simulink/Stateflow, where event passing/handling and synchronization is efficiently provided. We also employed Simulink Design Verifier toolset to prove correctness of the model with respect to given properties as well as some important properties from different components of the system. Initially, total property proving time was about ten hours on a Core 2 Due machine with 4GB of RAM. After applying optimizations such as removing deadlock dependencies in some states like *Distributer*, and specifying the precise proof assumptions to inputs, we could significantly reduce it to seven hours and sixteen minutes.

CHAPTER 4 ARTICLE 2
Formal Verification of Event-driven Health Applications

Mohammad-Reza Gholami, Hanifa Boucheneb
Journal of Software Engineering: Theories and Practices

Abstract– Software does an important task in almost every part of our everyday life, particularly in systems designed for the healthcare, aircraft navigation and automotive. One of the important objectives of software engineering is to support developers for building systems that function reliably even they are complex. In this paper, we show how model-based design is used to model a safety-related application. By applying formal verification techniques, we also define specific properties to ensure that a software system satisfies its correctness criteria. We use the formal approach to study and verify the properties of a medical device known as Endotracheal intubation. The system is modeled in a concurrent manner and synchronization between components is done through events. We present how the system is modeled in the Simulink and Stateflow and present formalization of some safety and temporal requirements based on the events issued from the controller. In order to formally prove the defined properties, we employ Simulink Design Verifier toolset.

Keywords– Formal Methods; Formal Verification; Design Verification; Model-Based Design; Linear Temporal Logic; Safety-Critical.

4.1 Introduction

The increasing complexity of embedded systems (e.g. avionics, health and automotive systems) conveys the producers of safety-critical applications to use more systematic processes for development. Traditional design processes are not fast enough in discovering the errors in requirements; hence, the whole process would be longer and more expensive.

A key goal of software engineering is to make it possible for developers to construct systems that operate reliably regardless of their complexity [27]. A promising approach to achieve this goal is to use formal methods, which are mathematically based languages, tools and techniques for specifying and verifying such systems. Formal methods can significantly increase our understanding of a system by disclosing inconsistencies, ambiguities, and incompleteness in the early design phase so that they can be eliminated in order to ensure the appropriate behavior of the system.

Software vendors typically spend large amounts of time of their total development time and resources on software testing [48]. The severity of the defects also increases the detection time and the cost of total development budget for a software product. So there is a need for a sophisticated discipline of software engineering to ensure that all the expected requirements were satisfied by means of the specified software. In other words, software verification refers to the process that can determine whether the artifacts of one software-development phase fulfil the specified requirements produced during the previous phase.

In a safety critical system, even a single error in the source code and an associated malfunction of the system can cause high costs and could even endanger the health of people. In critical real-time embedded systems, verification and validation activities are becoming huge and quite costly when the complexity and size of the systems grows.

Formal verification can be performed by employing Model-Based Design [33] in order to specify formal model of the system. Model-Based design facilitates the addressing of difficulties and complexities existing in control system design by providing an executable specification which implies that the model exists as more than just a document, but as a functional part of the design process [54]. It also provides a single design environment that enables developers to use a single model of their entire system for data analysis, model visualization, testing and validation, and ultimately product deployment, with or without automatic code generation [46]. Furthermore, model-based design creates a structure for software reuse that permits established designs to be effectively and reliably upgraded in a more simplistic and cost effective manner.

In this paper, we study how to verify some safety and liveness requirements of an event-based system using Simulink Design Verifier. We use the formal approach to verify properties of an Endotracheal intubation, which is a specific type of tracheal tube that is inserted through the patient's mouth to maintain an open airway [1]. This medical device is illustrated in Figure 4.1. Using Simulink/Stateflow, we come up with a model with parallel components, where event passing and synchronization is efficiently provided.

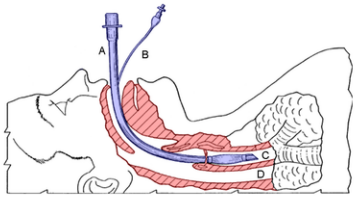


Figure 4.1 Endotracheal intubation [8]

The rest of the paper is organized as follows: Section 4.2 presents some background and previous work related to this research topic. Employed tools are described in Section 4.3. We present our case study in Section 4.4 which consists of definition, properties, and implementation of the model. The outcomes of our implementation are analysed in Section 4.5. Finally, Section 4.6 concludes our paper.

4.2 Background and Related Work

Formal methods enhance verification process by using formal notations and concepts in writing requirements and specifications. In formal methods, mathematical and logical techniques are used to express, investigate, and analyse the specification, design, documentation, and behavior of both hardware and software. The word *formal* in formal methods derives from *formal logic* and means "to do with form" [66]. In formal logic, dependence on human intuition and judgement is avoided in evaluating the arguments. In order to constitute an acceptable statement or a valid proof, formal logic employs a restricted language with very precise rules for writing assumptions, theorems, and proofs. In formal methods for computer science, languages are enriched with some of the ideas from programming languages and are called *specification languages*, but their underlying interpretation is usually based on a standard logic.

Formal verification in the field of software means the automated proof of specified properties on the code without executing the program. Also, it ensures that a design conforms to some precisely expressed notion of functional correctness [18]. The main benefits of formal verification in comparison to testing (dynamic verification), are its soundness and exhaustiveness. Specifications in formal methods are *well-formed* mathematical statements which are used to specify a property that needs to be verified in the system [10].

Many projects currently use MathWorks Simulink and Simulink Coder [9] which formerly known as Real-Time Workshop for at least some of their modeling and code development [29]. This kind of Design focuses on using executable system models as the foundation for the specification, design, implementation, test, and verification [13]. The executable specification in Model-based design, replaces parts or all of the paper format of the system specification and requirements as the main deliverable between design stages. It consists of an executable model of the application algorithm that can be simulated. The next step of Model-Based Design is known as model elaboration which consists of transforming the executable specification into a more design based form [60]. In Section 4.3, we briefly explain how to build an executable model by using Simulink and Stateflow.

Recently, some efforts have been made in order to employ formal methods in designing critical systems. In particular, Jiang *et al.* [42] developed a real-time Virtual Heart Model (VHM) for modeling the electro-physiological operation of proper functioning and malfunctioning. They introduced a timed-automaton model to define the timing properties of the heart and used Simulink Design Verifier as the main tool for designing their model.

Simulink/Stateflow has also been used in [32] to model a train tracking function for an automatic train protection system. The model was implemented based on the requirements specification document in which safety and functional properties were originally written in natural language. The authors of [32] used Simulink Design Verifier for verification and validation. They also had a positive experience when they used this tool for the safety-critical function in the railway transportation domain.

Another case example for a medical device has been presented in [57] where an iterative approach is applied for system verification based on software architectural model. They employed Simulink/Stateflow for describing the component level behavior of the model and used Simulink Design Verifier for proving the system level properties to establish component-level properties.

Above mentioned works used Simulink Design Verifier for proving safety properties which is supported by this tool in nature. In this paper, authors employ Simulink/Stateflow to model a system having different components. In addition, components of the system are designed to act in parallel and synchronization between them is accomplished by events. Furthermore, some liveness properties are also formalized in the model to describe our method for proving this kind of properties. For the formal analysis, we use Simulink Design Verifier, which intensively employs the BMC and K-Induction features of the PROVER[2] engine to establish the satisfiability of the proof objectives. We also use this tool to verify design issues like *Integer overflow*, *Division by zero*, *Assertions and Violations of design properties* to eliminate runtime issues of the model.

4.3 Simulink and Stateflow

The executable model can be built by Simulink which is an environment for multi-domain simulation and Model-based Design for dynamic and embedded systems. Mode logic in Simulink models is described in terms of hierarchical state machines specified in a variant of Statecharts called Stateflow [9].

Stateflow is a widespread model-based development environment is Matlab/Simulink toolset, which is used in several industries, such as aerospace, medical, and automotive. It uses a

variant of the finite state machine notation established by Harel [38] and provides the language elements required to describe complex logic in a natural, readable, and understandable form. Since it is tightly integrated with MATLAB and Simulink, it can provide an efficient environment for designing embedded systems that contain control and supervisory. In particular, Stateflow diagram enables the graphical representation of hierarchical and parallel states as well as transitions between them and inherits all simulation and code generation capabilities from Matlab toolset.

A state is called as *superstate* when it contains other states and a state is called *substate* when it is contained by a supersate. When a state consists of one or more substates, it has *decomposition* that can be either parallel (AND) or exclusive (OR) decomposition. All substates at a particular level within the same state must have the same decomposition.

In parallel (AND) decomposition, states can be active at the same time and the activity of each parallel state is essentially independent of other states.

We can use our defined *Events* to trigger actions in parallel states of a Stateflow chart. Broadcasting of an event can trigger a transition and/or an action. The *actions* can be executed either as a part of a transition from one state to another or based on the activity status of a state which can be entry, during, exit, and on event actions. For instance, while the state *Fill* is active, $c = GetElapsed()$ is executed every time unit.

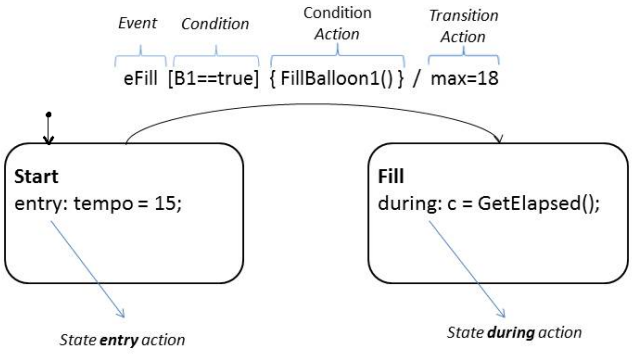


Figure 4.2 Stateflow Semantics

The general form of a transition in Stateflow is presented in Figure 4.2. It shows the behavior of a simple event, condition and transition action specified on a transition from one exclusive (OR) state to another. Initially, state *Start* is active and *entry* action is executed, which sets the variable tempo to 15. When the event *eFill* is received, the chart root detects that there is a valid transition to state *Fill* as a result of the event *eFill*, so it validates the condition and if the result is true, the *Condition Action* immediately gets executed and completed.

Conversely, the state *Start* remains active and no *Condition Action* executes if the condition is false. The state *Start* is marked as inactive and the *Transition Action* is executed and completed when the transition destination *Fill* has been determined to be valid. States can have different actions such as: *entry*, *during*, *exit*, and *on event-name* which are being executed based on the current status of the active state.

4.3.1 Simulink Library

To build models in Simulink, blocks are the main elements that are used, and they are hosted in the library. A Simulink block has sets of *input* and *output* ports. A block with N input and M output ports defines a function which describes each of the signals at the output ports as a (possibly time-dependent) expression of the signals at the input ports. Formally, a block is a tuple (P_i, P_o, f), where P_i is the set of input ports, P_o is the set of output ports and $f : \mathbb{R}^{\mathbb{N}} \rightarrow \mathbb{R}^{\mathbb{M}}$ is a function which defines the behavior of the block [52]. In the following, some blocks from the standard Simulink library is briefly described:

Embedded Matlab Function block

The Embedded MATLAB Function Block facilitates writing the MATLAB m-code which can be incorporated into a Simulink model. This block is placed in the User Defined Functions Library and can be inserted into a model in the same way as any other Simulink blocks. We use this block whenever there is a need to implement part of the logic of the property by program code.

Subsystem block

A subsystem is a set of blocks that we replace with a single block called a Subsystem block. As our model increases in size and complexity, we can simplify it by grouping blocks into subsystems.

Function-Call Subsystem

This block represents a subsystem that can be invoked as a function by another block. In other words, a function-call subsystem is a subsystem that another block can invoke it directly during a simulation. It is similar to a function in a procedural programming language. Invoking a function-call subsystem is equivalent to invoking the output methods of the blocks that the subsystem contains in sorted order. The block that invokes a function-call subsystem

is called the function-call initiator. Moreover, the Stateflow, Function-Call Generator, and S-function blocks can all serve as function-call initiators.

Temporal Logic Operators

These operators are used in the Stateflow and control the execution of a chart in terms of time. In state actions and transitions, you can use two types of temporal logics: event-based and absolute-time. Event-based temporal logic keeps track of recurring events, and absolute-time temporal logic defines time periods based on the simulation time of your chart.

For event-based temporal logic, the following operators *after*, *before*, *every* and *temporalCount* can be used in the Stateflow. For instance, in the provided model for our case study, the operator *after* is used in *Timers* and *Cylinders* states.

4.3.2 Simulink Design Verifier

Simulink Design Verifier® [5] is a tool set of Matlab which uses *formal methods* to identify hard to find *design errors* in the models without requiring extensive tests or simulation runs. Moreover, it enables us to perform model analysis within the Simulink environment, in order to verify the designs and validate the requirements early, without having to generate code.

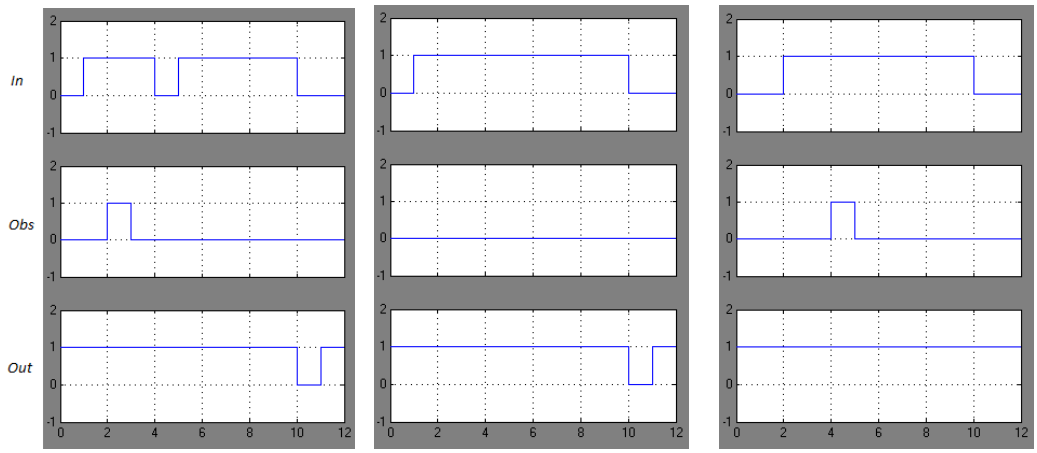
We can use Simulink, MATLAB functions, and Stateflow to express formal requirements. It also provides a set of building blocks and functions that can be used to define and organize verification objectives. The block library provided, includes blocks and functions for test objectives, proof objectives, assertions, constraints. In addition, a dedicated set of temporal operators like *Detector*, *Extender* and *Within Implies* blocks are also provided in order to model the verification objectives with temporal aspects. Following to this, *Within Implies* block that is used in our implementation is briefly described:

Within Implies block

The Within Implies block captures the within implication by observing whether the *Obs* input is true for at least one step within each true duration of the first input *In*. Whenever *Obs* is not detected within a particular input true duration, the output becomes false for one time step in the step that follows the input true duration. This block captures the behaviour: ('Within' *In*) $\Rightarrow$ *Obs*.

In the example illustrated in Figure 4.3, model sample time is considered as 1 second.

— In Figure 4.3a, although *Obs* is observed within the first true duration of *In* (steps



(a) False Output (b) False Output (c) True Output

Figure 4.3 Within Implies Simulink Block

1...4), but it is not observed within the second true duration of *In* (steps 5...10), so *Out* becomes false for one time step after the *In* signal becomes false (step 11).

— In Figure 4.3b, *Obs* is not observed within the first true duration of *In*, so *Out* becomes false for one time step after the *In* signal becomes false.

— In Figure 4.3c, *Obs* is observed within the true duration of *In* (at time step 4), so *Out* remains true until the end of the simulation. The input is true if Obs becomes valid for at least one time every true duration of the input.

4.4 Case Study

This case study aims to show and familiarize how a system works with a framework where time aspects are combined with multi task programming. In this case study, we are modeling and verifying the properties of a Filling system of balloons of an intubation probe. An intubation probe is placed to ensure continuous passage of air to the lungs and introduce oxygen sensors, aspiration probes to the lung for patient treatment.

As illustrated in Figure 4.4, this system consists of two balloons, two access valves for manual inflation, two pressure sensors, a power distributor, a pump and an air tank. The pump is actuated by a gear motor and a transmission by a cylinder rack. The pumped air is propelled through the power distributor (B and D) to one of the balloons or outside. The probe has several buttons (*Start*, *Stop*, *Duration*, *Pressure*, *StopAlarm*) and LEDs (*L1*, *L2*, *Alarm*). The Alarm LED reports the anomalies. L1 and L2 are witnesses indicating the inflated balloons, and the button *StopAlarm* allows the user to stop the alarm. The system controlled by

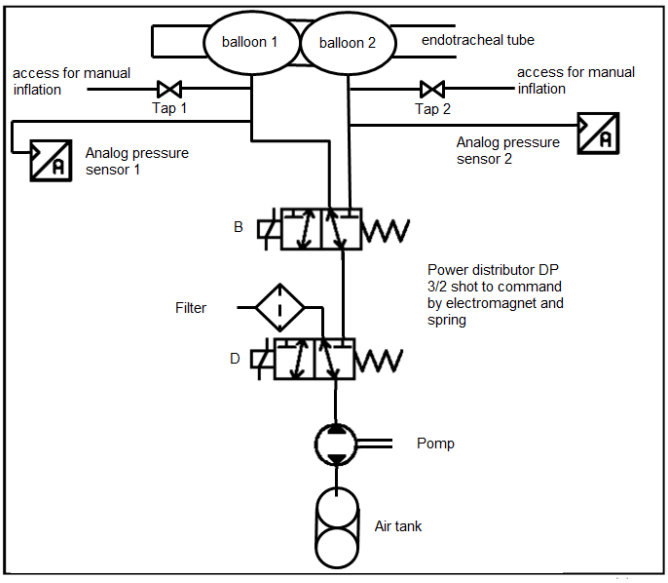


Figure 4.4 Physical Features of the control system

a Programmable Controller who is responsible for controlling the commands and messages which are sent to or received by other components. The controller operation is also described by means of the Grafcet [73] that is shown in Figure 4.5. We are using Simulink and Stateflow as an integrated tool environment for modeling, and Simulink Design Verifier for verification of some properties.

4.4.1 The model

A model is known as abstract representation of a system. Software model is actually the ways of expressing a software design, and in order to express the software design some kind of abstract language or pictures are usually used. Software modeling needs to deal with the entire software design, including interfaces, interactions with other software, and all the software methods [34]. Engineers can model the system using a modeling language which it can be graphical or textual [39].

According to the description of the case study, the first step when modeling the Intubation, is to pinpoint the superstates in the system and their interactions. One of the most important parts of the design is to find out which superstates should be parallel (AND) and which ones should be exclusive (OR).

In the Intubation Stateflow, there are ten distinguished blocks corresponding to each component, which are illustrated in Figure 4.6. All of these blocks are working in a parallel execution order. These blocks are represented with ten different states in the model. In ev-

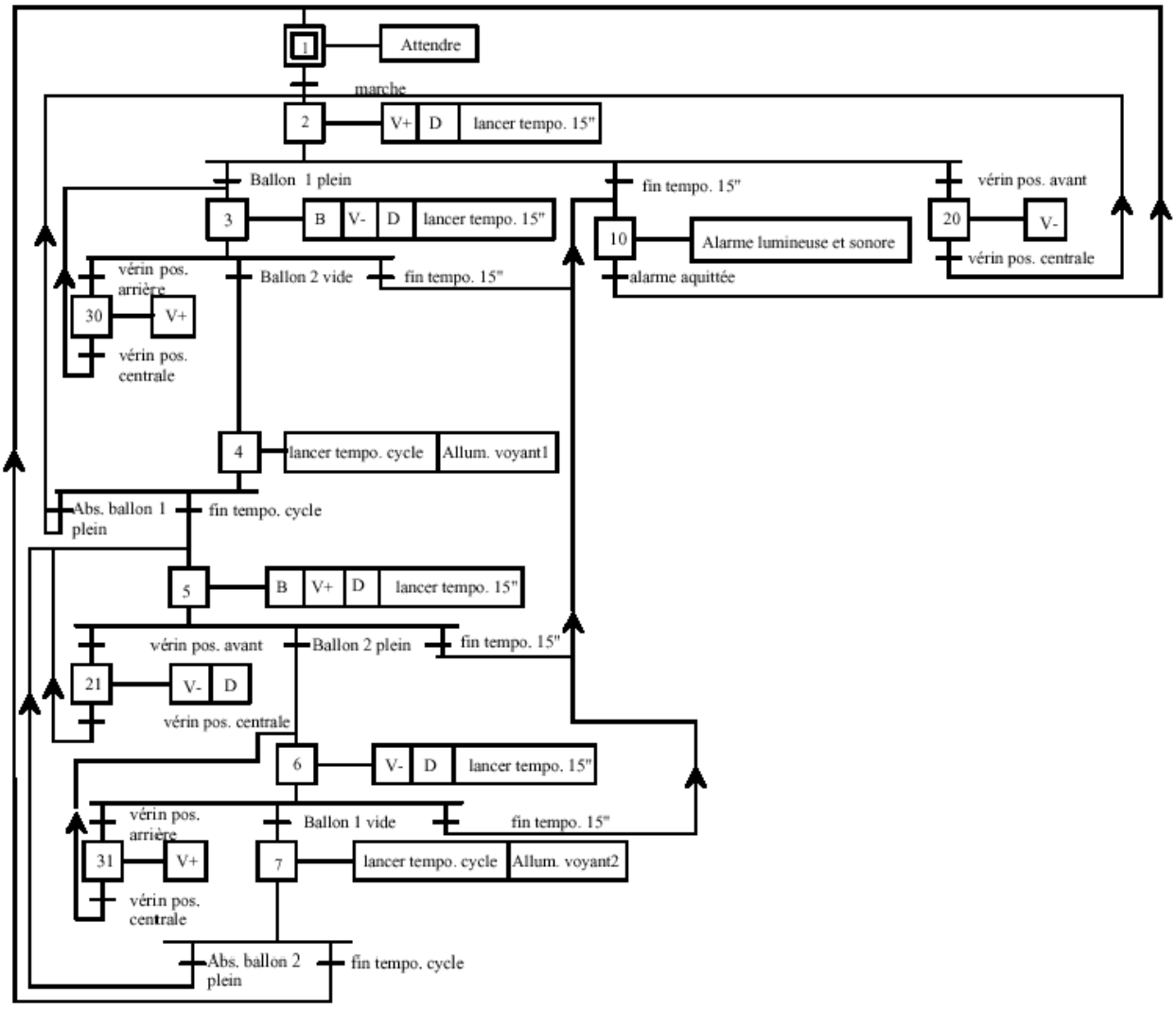


Figure 4.5 Grafcet describing the operation of the controller

ery moment of running the model, at least one substate has to be active in each state. These states are: *Controller*, *Distributor*, *Cylinder*, *Balloons*, *Alarm*, *Lights* and *Timers*. These states are designed to be parallel (AND) because a change in these states is allowed at every time step.

The states corresponding to each component, interact together through sending direct broadcast event and one simplified function (*Initialize*) to make the model smaller, initialize variables and the status. In other words, using function helps to group the different actions that are associated to each transition. Direct event broadcasting is used to prevent receiving an error pertaining to recursion in the Stateflow chart. The temporal logic operator *after* is also used whenever there was a need to control the execution of states in terms of time. In

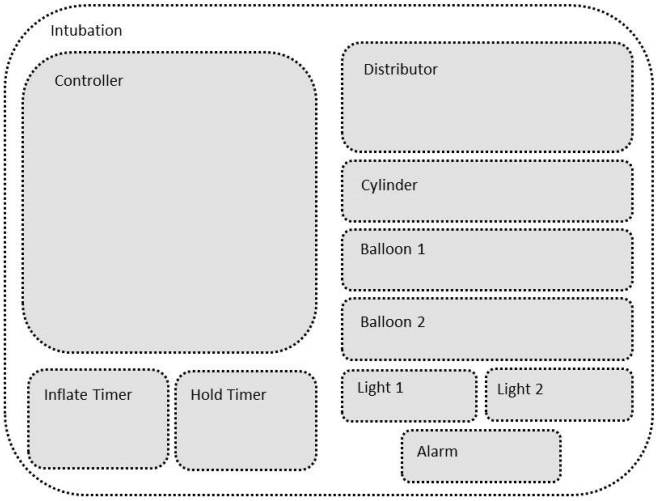


Figure 4.6 Components of the Endotracheal Intubation

figures, some defined functions in the Stateflow are removed for the sake of simplicity. To explain and describe the model, this section is divided into three different parts as follows:

- Inputs & Local variables, ranges and default values.
- Events.
- Components (Parallel (AND) States).

In the following, we give a short description of components, including some screenshots taken from the Simulink to complement the description of the Stateflow.

Variables:
We maintain the current state and values of the components using local variables. For simplification, we used integer values to represent the corresponding physical step of the components. The input type variable corresponds to the variable whose value is coming from the Simulink model. Conversely, the output type variable is modified and used in the Stateflow, and it is accessible through the Simulink model. Both input and output type variables, their range, and default values are defined and set in the declaration of the stateflow, for example, for the cylinder component:

$int\ nCylPos = -1$

For this specific variable, the value can be set to either -1, 0 or 1 which respectively correspond to the position of the cylinder as: *rear*, *center* and *front*. Similarly, the different values for $nBallonState$ correspond to different status of the balloon, such as: *Empty*, *NotFull* and *Full*. The target pressure of the balloon in which the balloon is considered as full inflated,

is stored in *nPressure*. The variable *nDuration*, sets the amount of the time that an inflated balloon should remain full before controller sends a deflation command.

Some input and output variables for the Stateflow diagram listed in Table 4.1.

Table 4.1 Variables in Intubation Stateflow

Name	Type	Values
nCylPos	Output	-1, 0, 1, 2
nBallonState	Output	-1, 0, 1
bLightState	Output	true, false
bAlarm	Output	true, false
nDuration	Input	10, 20, 30 mins
nPressure	Input	12, 18, 24
bStopAlarm	Input	true, false
bStart	Input	true, false

Events:

Different events were defined to model the communication between different components of the system. According to their usage, they are defined in the Stateflow as *Directed Event Broadcast*. The relationship between events and the corresponding component, is listed in the Table 4.2. The status column illustrates if the event is sent to components of the chart or is sent out from the Stateflow to the Simulink model of the system. As such, the event *exFillB1* means that the event will be sent out from *Distributer* component whenever a Fill attempt is required to inflate balloon 1. Similarly, the external event *exEmptyB1* is sent out from *Distributer*, for every deflating requests. In addition, the event *exFillB1* is also sent from *Balloon1* component out from the Stateflow whenever balloon 1 is completely inflated.

Table 4.2 Events

Component	Events	Status
Alarm	eStartAlarm, eStopAlarm	Internal
Light1 Light2	eLight1On, eLight1Off, eLight2On, eLight2Off	Internal Internal
Distributor	eFill, eEmpty, exFillB1, exFillB2, exEmptyB1, exEmptyB2	Internal External External
Cylinder	vPlus, vMinus	Internal
Balloons	exB1Full, exB2Full	External

Stateflow:

This section describes each component that we have modeled as different states. We explain

the nature and the interaction of each one of them.

Controller

The model of the controller was realized from the specification based on the provided Grafcet illustrated in Figure 4.5. In order to model the controller some local variables representing the steps, transitions and actions and some local variables representing the channels and reflecting the value of local actions in the system were also defined. The local variables *B* and *D* are defined as *boolean*, and being used to set the channels and activate them (B and D are shown in Figure 4.4).

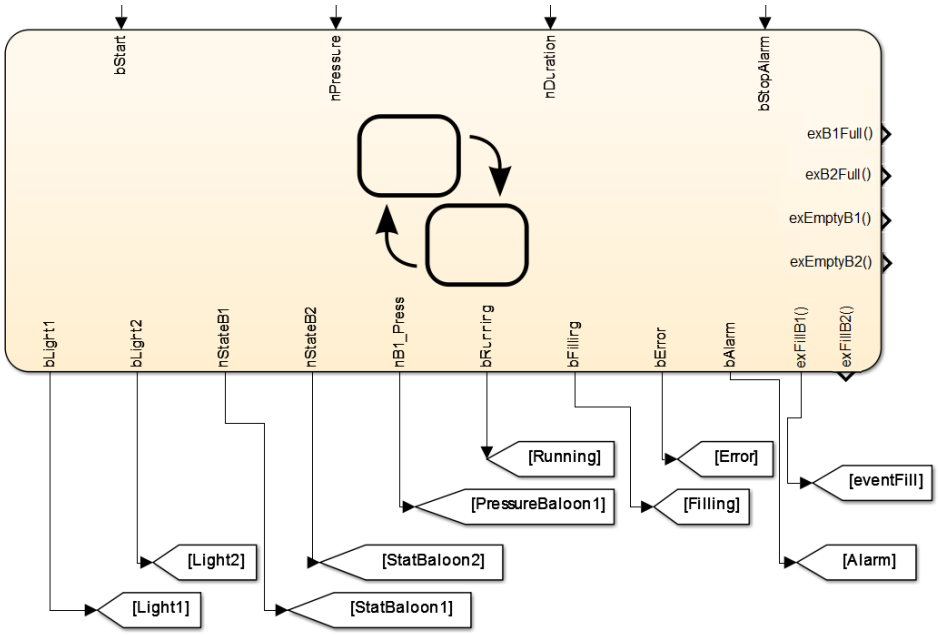


Figure 4.7 Intubation Stateflow with its I/O

The entire controller is included in a single superstate. In order to do a specific action, the controller set the values for the channels and send the event to the *Distributor* state. Table 4.3 presents events and channels used by the controller for specific actions.

Instead of modeling the user's interaction with the system, we use local variables to simulate the choice of target pressure and the targeted cycle time (*nPressure* and *nDuration*). Those values are set at the beginning and are left untouched during the simulation. Our model assumes that at the beginning the cylinder is in the position *rear* and both balloons are considered *Empty*.

One of the major building blocks of our controller which is responsible to inflate and deflate the balloons is illustrated in Figure 4.7. In this figure, in order to inflate the first balloon, on the entry action of the state *S2* corresponded values for B and D are set then the event *eFill* is sent to the *Distributor*.

Cylinder

The cylinder has three substates: *rear*, *center* and *front*. This state is constantly waiting to receive the events *vPlus* or *vMinus* from the controller to change its position forward or backward. To model the 2 seconds delay for each position transition, we included an in-between location, and used an *after* temporal operator between each position. Once the delay is exhausted, the cylinder position changes. We use a local variable *nCylPos* (*rear* = -1, *center* = 0, *front* = 1 and *In-between* = 2) to store the current position of the cylinder. Initially, the state *rear* is active. We set this variable to 2 when the cylinder is in transition between two positions. The controller has guards using the transitional value to ensure the cylinder state has completed its movement.

Pressure Distributor

This state receives the specified event from the controller in order to launch the selected action for inflating or deflating the desired balloon. The selected action is relevant to the current value of the local variables B and D which are set to *true* or *false* by the controller before sending the event *eFill* or *eEmpty*. In addition, to complete the entire function, it also sends specific events to states *Cylinder* and *Balloon* for their relevant actions. Table 4.3 shows the events and variables used for specific functions.

Table 4.3 Actions

Distributor Channel	**Event**	**Function**
B=0, D=1	vPlus	Inflate ballon 1
B=1, D=0	vPlus	Inflate ballon 2
D=0 or B=0, D=0	vPlus	Move cylinder forward
B=0, D=1	vMinus	Deflate ballon 1
B=1, D=1	vMinus	Deflate ballon 2
D=0 or B=0, D=0	vMinus	Move cylinder backward

Initially, the state *Init* is active and waits to receive a specific event from *Controller* to complete the selected function. The state *FillB1*, on the rightmost side of Figure 4.8 has to

run iteratively until the destined balloon is inflated. Similarly, the emptiness of the balloon is ensured by running the state *EmptyB1* on the leftmost side of this figure.

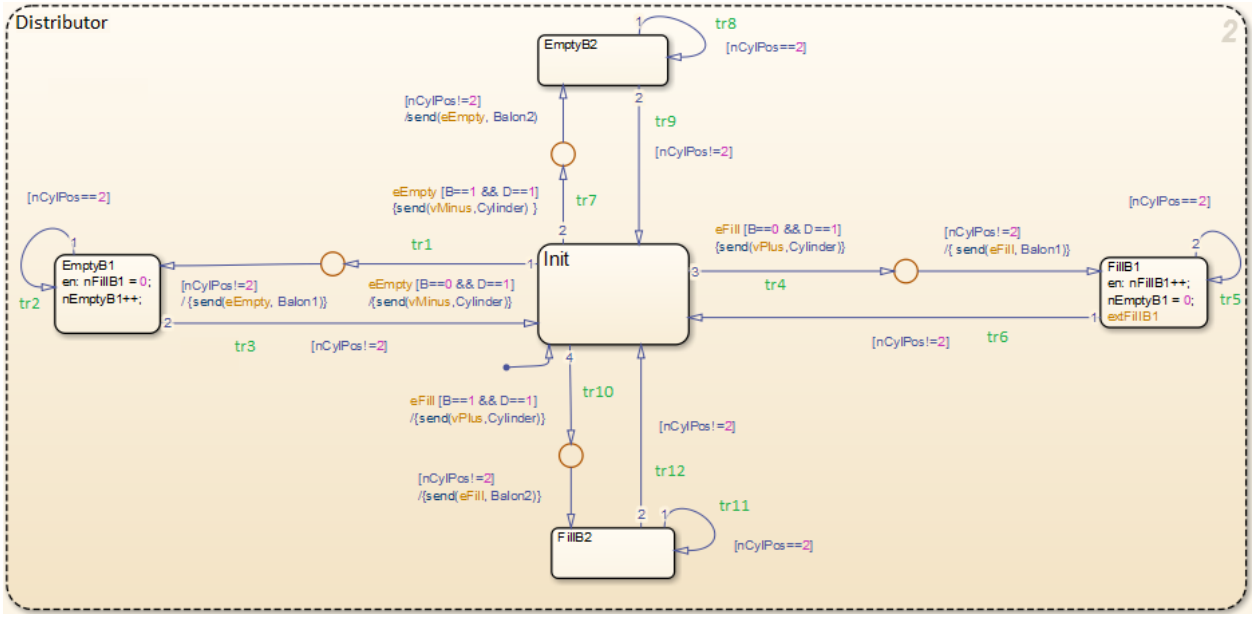


Figure 4.8 Pressure distributor

The state of this component is formally defined by sets of states, actions and transitions. First thing to remember is that the composition of states can be either an *Parallel AND* or an *Exclusive OR* composition.

Outgoing transitions for *Distributor* state as illustrated in Figure 4.8, are defined as:
$TR = \{tr_1, tr_2, tr_3, tr_4, tr_5, tr_6, tr_7, tr_8, tr_9, tr_{10}, tr_{11}, tr_{12}\}$.

A state definition *sd*, is a triplet composed of actions $A = \{\mathbf{e}ntry, \mathbf{d}uring, e\mathbf{x}it\}$, executed respectively upon entering, during, and exiting the state, an internal composition (Parallel AND, Exclusive OR), and a list of outgoing transitions [37].

The state definition list *SD* [37], which associates state definitions *sd* with corresponding substates in the *Distributor*, is denoted as:

$$SD = \{Dist : sd_0; Init : sd_1; EmptyB1 : sd_2; FillB1 : sd_3; EmptyB2 : sd_4; FillB2 : sd_5\}$$

The definition of the states for the *Distributor* depicted in Figure 4.8, can be listed as:

- $sd_0 = (AND)$
- $sd_1 = (OR, \{tr_1, tr_4, tr_7, tr_{10}\})$
- $sd_2 = ((A.e), OR, \{tr_2, tr_3\})$

— $sd_3 = ((A.e), OR, \{tr_5, tr_6\})$
— $sd_4 = ((A.e), OR, \{tr_8, tr_9\})$
— $sd_5 = ((A.e), OR, \{tr_{11}, tr_{12}\})$

The sample trace information represented in Table 4.4, illustrates the status of events and evolution of states over the time, when the controller tries to inflate the first balloon which is empty. Values of B and D channels specifies the current requested function to the pressure distributor component. During the process some events are handled or triggered by *Distributor* component and is denoted as follows: $Event_{(Recv)}$ is an internal event that is issued from the controller to distributor component to tell which of these functions (Inflate, Deflate, Move Cylinder) is requested. $Event_{(Send)}$ is an internal event that will be sent from *Distributor* state to *Cylinder* state to perform the movement. $Event_{(Extern)}$ is sent from *Distibutor* state out from the Stateflow chart. This event later will be handled by other subsystem block in the Simulink model.

Balloons

The state balloon has three substates: *Empty*, *NotFull* and *Full*. In this model, we consider two different superstates corresponding to each balloon. Initially, the state *Empty* is active in both balloons. The transition between substates has a guard; so, the movement is done when one of the events *eFill* or *eEmpty* is received from the state *Distributor*. We use a local variable *nBalloonState* (*Empty* = -1, *NotFull* = 0, *Full* = 1) to store the current status of the balloon.

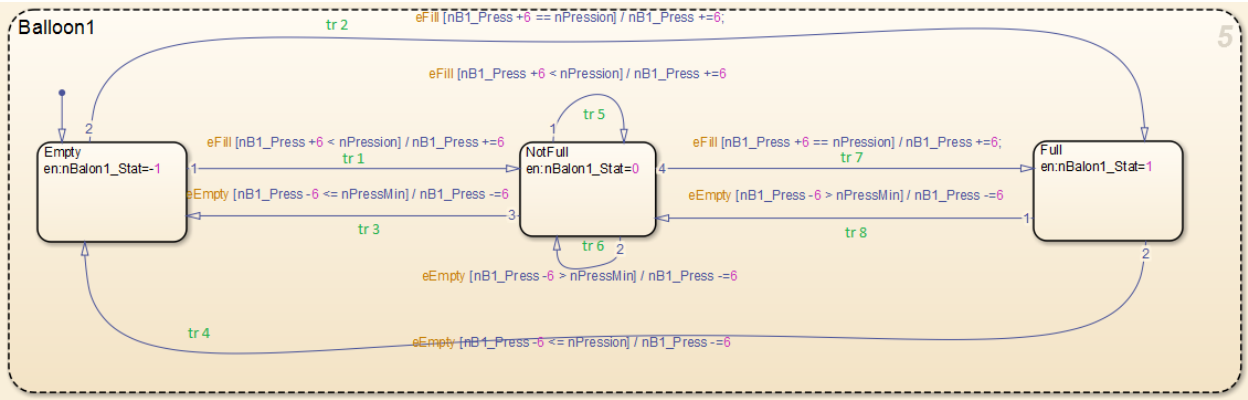


Figure 4.9 Stateflow of Balloon1

Outgoing transitions for *Balloon1* state as illustrated in Figure 4.9, are defined as:
$TR = \{tr_1, tr_2, tr_3, tr_4, tr_5, tr_6, tr_7, tr_8\}$.

Table 4.4 Distributor - Evolution of states and status of events over the time

t	t_0	...	t_i	t_{i+1}	t_{i+2}	t_{i+3}	t_{i+4}	t_{i+n}
B	0	0	1	1	1	1	1	0
D	0	0	0	0	0	0	0	0
S_a	$Init$	$Init$	$Init$	$Init$	$FillB1$	$Init$	$FillB1$	$Init$
$Event_{(Recv)}$	-	-	$eFill$	-	$eFill$	-	-	-
$Event_{(Send)}$	-	-	$vPlus$	-	$vPlus$	-	-	-
$Event_{(Extern)}$	-	-	-	-	$eFill(B1)$	-	$eFill(B1)$	-
$CylPos$	$rear$	$rear$	$rear$	$rear$	$center$	$center$	$front$	$front$
$Baloon1$	$empty$	$empty$	$empty$	$NotFull$	$NotFull$	$full$	$full$	$full$

The state definition list *SD*, which associates state definitions *sd* with corresponding substates in the *Balloon1*, can be denoted as:

$SD = \{Baloon1 : sd_0; Empty : sd_1; NotFull : sd_2; Full : sd_3\}$

The definition of the states for the Stateflow depicted in Figure 4.9, can be listed as:

- $sd_0 = (AND)$
- $sd_1 = ((A.e), OR, \{tr_1, tr_2\})$
- $sd_2 = ((A.e), OR, \{tr_3, tr_5, tr_6, tr_7\})$
- $sd_3 = ((A.e), OR, \{tr_4, tr_8\}\})$

Timers

In our model, timers are designed as two different components: The *InflateTimer* which is responsible for the inflation time of a balloon, and the *HoldTimer* which is the time that a balloon should maintain the status *Full*.

The state *InflateTimer* contains three substates: State *Wait* which is initially active, the state *Start* which is activated by the controller while requesting for a *Inflate* procedure, and *Stop* that gets activated whenever the maximum time is reached. The temporal logic operator *after* is used as a transition condition from state *Start* to state *Stop*. The *InflateTimer* is illustrated in Figure 4.10:

Outgoing transitions for the *inflateTimer* state as illustrated in Figure 4.10, are defined as: $TR = \{tr_1, tr_2, tr_3, tr_4\}$.

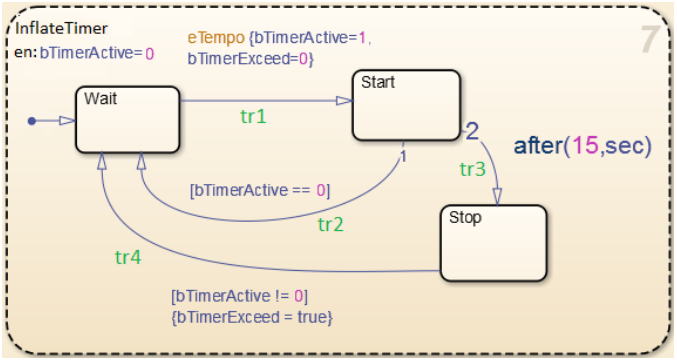


Figure 4.10 Inflate/Deflate Timer State

The state definition list SD, which associates state definitions sd with corresponding substates in the *inflateTimer* can be denoted as:

$$SD = \{inflateTimer : sd_0; Wait : sd_1; Start : sd_2; Stop : sd_3\}$$

The definition of the states for *inflateTimer* that is depicted in Figure 4.10, can be listed as:

- $sd_0 = ((A.e), AND)$
- $sd_1 = (OR, \{tr_1, tr_2, tr_4\})$
- $sd_2 = (OR, \{tr_1, tr_2, tr_3\})$
- $sd_3 = (OR, \{tr_3, tr_4\})$

Alarm

The alert state contains two substates: *Off* and *On*. Initially, the state *Off* is active. When the timer exceeds from the specified threshold or if any anomaly happens, the controller stops the system operation and sends the event *eAlarm*. Once the event received, the state *On* will be activated and remains in this state until the user stops the alarm.

4.5 Methods and Analysis

This section describes our method and results of formal verification using Design Verifier with Simulink and Stateflow. Before verification, we run the simulation for our provided model using predefined input parameters in order to ensure that the model can be executed properly.

4.5.1 Properties

The term property refers to a logical expression of signal values in a model. For example, we can specify that a signal in a model should attain a particular value during execution of the system. The Simulink Design Verifier software can then prove the validity of such safety properties. This is done by performing a formal analysis of the model to prove or disprove the specified properties. If the software disproves a property, it provides a counterexample that demonstrates a property violation.

The developer can specify properties by using two blocks provided in the Simulink Design Verifier library. The *Proof Objective* block is used to define the values of a signal that the Simulink Design Verifier software will prove. The *Proof Assumption* block is used to constrain the values of a signal during a proof [39].

The definition of properties comes with the execution order of contained blocks (e.g. a, b, c, ...). The update functions of each block in the property gets executed respectively after other blocks in the main model. The modeled system consists of the following properties:

1. A balloon should be inflated in less than or equal three fill attempts.
2. The fill command remains active until corresponding balloon is inflated.
3. There must be no anomaly alarm (False alarms).
4. The pressure in each balloon never exceeds a predetermined value.

The term $Pre(x,t)$ is used for *Predecessor* in provided equations when defining specified properties (It corresponds to *Unit Delay* block in Simulink library). As such, The predecessor of signal x at time t is denoted by $Pre(x,t)$ that corresponds to the value of that signal at time step $t-1$. The term $Pre^*(x,t)$ is also used as the transitive closure of $Pre(x,t)$. The following section details each property and the results obtained by Simulink Design Verifier:

Property 1

The goal of this property is to verify the number of fill attempt events that are sent by the controller, in an inflate procedure, meets the number specified in the requirement specifications. Consider the scenario that the controller needs to inflate balloon 1 at the beginning of the system run. At start, we assume that the balloon is empty (has 6 cm of water) and the cylinder is at *rear* position, and each *Fill* attempt increases 6 cm of water to the balloon's pressure. If the target pressure is set to 18 cm of water, the controller needs to issue 2 fill attempts in the inflation procedure.

Figure 4.11, illustrates the implementation of this property in Simulink. The *Distributer* component sends out an external event from the Stateflow whenever a fill attempt is issued.

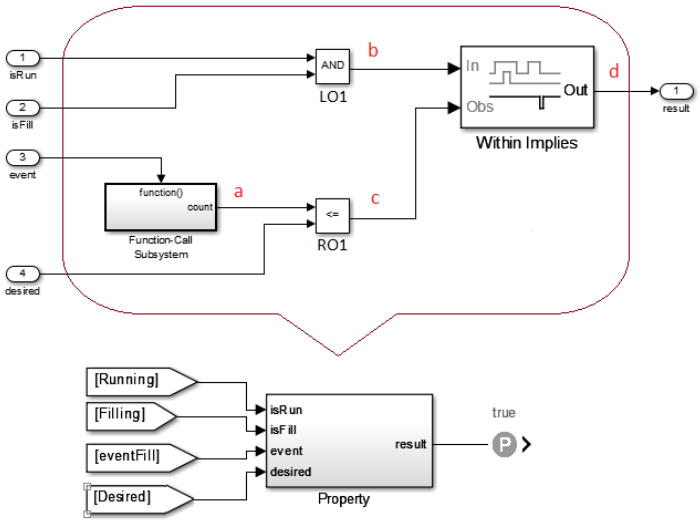


Figure 4.11 Formalization of property 1

The *Function-Call* Subsystem is a triggered subsystem and increases the count of the attempts whenever an event is received.

In Figure 4.11, the Logical Operator *LO1* validates *isFill* and *isRun* input signals, and ensures that 'b' becomes *false* as soon as the inflating procedure is terminated. The Relation Operator *RO1* validates the count input against the desired value. Afterwards the *Within Implies* block checks if the input 'c' was true for at least one time step during the time steps that 'b' is true.

In Figure 4.12, as per definition of *Within Implies* block shown in Equation 4.4, it captures the within implication by observing whether the input $'c'$ is *true* for at least one time step within each true duration of the first input $'b'$.

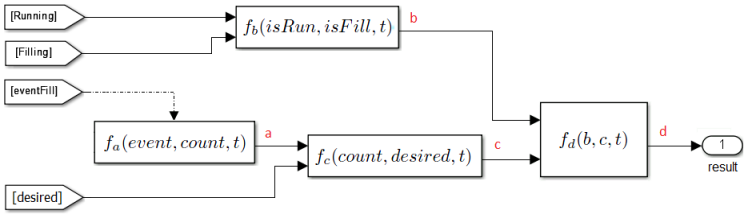


Figure 4.12 Internal functions in property 1

Formalization: Let $\mathbb{M}$ be a Simulink model, and property 1 denoted by p:

$\mathbb{M} \models p \quad$ iff $\quad G\ f_d(b, c, t) = 1$

Proof: The block function of this property can be written as: $f_d(b, c, t)$, where the output should hold *true* during the execution for any time t.

As illustrated in Figure 4.12, this function can be divided into four different sub-functions that are denoted as:
$f_a(event, count, t)$,
$f_b(isRun, isFill, count, desired, t)$,
$f_c(count, desired, t)$,
$f_d(b, c, t)$.

Let n be the number of consecutive time steps that b is true. The output of each function over the evolution of time t is as follows:

$$f_a(event, count, t) = \begin{cases} 0 & \text{if} \quad t = 0 \\ count + 1 & \text{if} \quad t > 0 \ \wedge \ event = 1 \\ count & Otherwise \end{cases} \tag{4.1}$$

$$f_b(isRun, isFill, t) = \begin{cases} 1 & \text{if} \quad (isRun \wedge isFill) \\ 0 & \text{if} \quad \neg(isRun \wedge isFill) \end{cases} \tag{4.2}$$

$$f_c(count, desired, t) = \begin{cases} 1 & \text{if} \quad count = desired \\ 0 & \text{if} \quad count \neq desired \end{cases} \tag{4.3}$$

$$f_d(b,c,t) = \begin{cases} 0 & \text{if } \neg b_{(t)} \wedge \neg Pre^*(c,t) \wedge (t = n+1) \\ \\ 1 & Otherwise \end{cases} \tag{4.4}$$

Finally, d as the output of this property, must hold *true* during the entire execution of the system and corresponds to the result of $b(t) \Rightarrow c(t)$.

Simulation and trace of the property p is done by specifying the time offset $T_o = 0$ and sample time period $T_s = 1sec$. In the following, the active state in *Cylinder* component as well as output variables of the Stateflow are shown for these situations:

— If cylinder is at rear position.
— If cylinder is at center position.

Table 4.5, denotes the evolution of variables over the time when the cylinder is at *rear* position. We assume that the inflate procedure for balloon 1 is started at time step t_i by the controller. Initial values are also shown in the table at time 0.

As per design of the controller, when a *fill* command is issued but if the cylinder position is in *front* position, the controller holds the *fill* command, sends backward command to the cylinder and re-issues the fill command and sends the external *event*. Table 4.6, displays the evolution of the variables over the time when the cylinder is at *center* position.

In terms of block execution sequence, the execution order of *Implies* block is after 'Intubation' chart, and *Proof Objective* block is after the *Implies* block. Thus, in each time step of the execution, inputs of the Stateflow are evaluated first and then its outputs are updated and fed to the specified property including the *Implies* block.

The implementation of the function f_a in Figure 4.12 is illustrated in Figure 4.13.

To explain, the purpose of using the *Unit Delay* block is to prevent causing the *Algebraic-loop*. As depicted in Figure 4.13 the evaluation of the *Sum* block is done in each sample time, so direct feedback the output of the Sum block to one of its inputs causes an algebraic loop and reports an error in compile time, so there is a need to use *Unit Delay* block which has an internal state and stores the previous input. The initial state of this block is set to zero.

We also proposed another implementation for the function f_a to investigate the verification time of this property when two different implementations are used. To address this, we

Table 4.5 Evolution of variables over the time - Cylinder at rear

t	0	...	t_i	t_{i+1}	t_{i+2}	t_{i+3}	t_{i+4}	t_{i+5}	t_{i+6}	...
Cylinder	*rear*	*rear*	*rear*	-	*center*	-	*front*	-	*center*	...
isRun	0	1	1	1	1	1	1	1	1	...
isFill	0	0	1	1	1	1	1	0	0	...
event	-	-	-	-	1	-	1	-	-	...
count	0	0	0	0	1	1	2	2	2	...

Table 4.6 Evolution of variables over the time - Cylinder at center

t	0	...	t_i	t_{i+1}	t_{i+2}	t_{i+3}	t_{i+4}	t_{i+5}	t_{i+6}	t_{i+7}	t_{i+8}	...
Cylinder	*center*	*center*	*center*	-	*front*	-	*center*	-	*front*	-	*center*	...
isRun	0	1	1	1	1	1	1	1	1	1	1	...
isFill	0	0	1	1	1	1	1	1	1	0	0	...
event	-	-	-	-	1	-	-	-	1	-	-	...
count	0	0	0	0	1	1	1	1	2	2	2	...

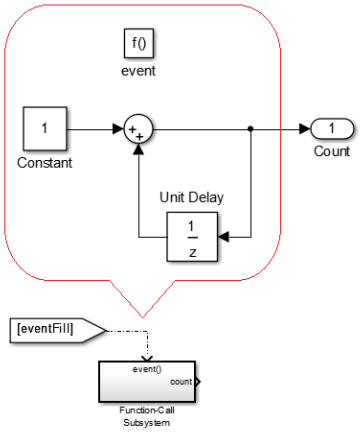
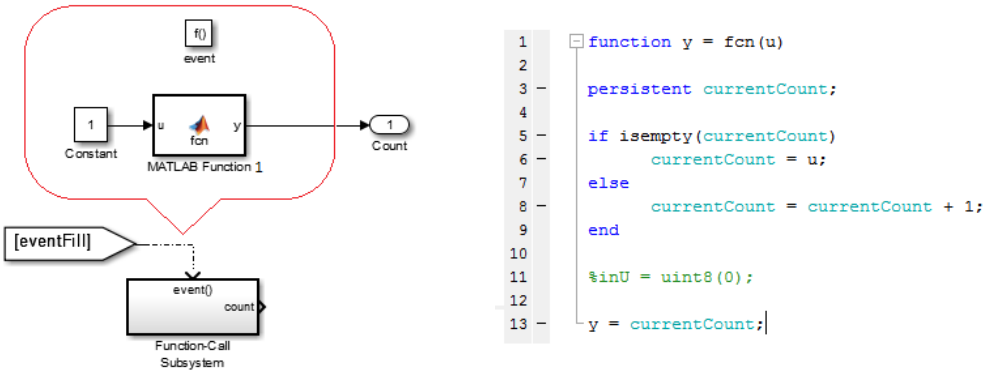


Figure 4.13 Using Standard Simulink Block

employed an Embedded Matlab function which has the same functionality and increments the output *count* as soon as receiving the external event from the controller. Figure 4.14a represents the subsystem's internal blocks along with its corresponding code.



(a) Embedded Matlab Function (b) Subsystem Code

Figure 4.14 Function-call Subsystem Block

Property 2

The goal of this linear temporal property is to make sure when a controller initiates an inflation procedure, the fill signal remains enable until the corresponding balloon is inflated completely. As per design of the controller, when the state corresponding to the inflation of first balloon is activated, it activates the Filling signal and send the appropriate events to the *Distributor* and *Cylinder* respectively in order to complete the inflating process. Since in each movement of the *Cylinder* component adds a specific pressure (6 cm of water) to the balloon, this sequence should be executed more than one time, based on the target pressure that is defined for the balloons in the configuration (18 cm of water for an inflated balloon).

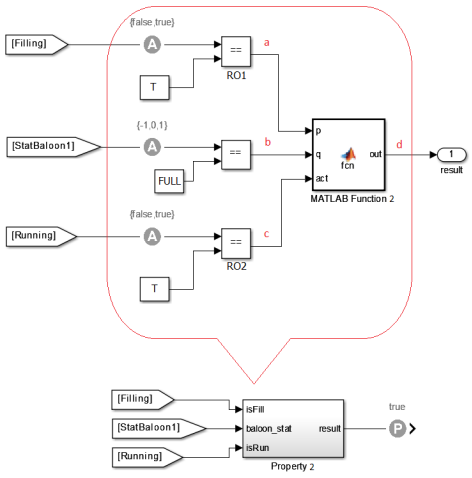


Figure 4.15 Property 2

Figure 4.15, illustrates the implementation of this property. The input *Running* is used to bound the time steps of the verification to the time that the system is running.

As illustrated in Figure 4.16, this function can be divided into four different sub-functions that are denoted as:

$f_a(Filling, t)$,
$f_b(baloon\_stat, t)$,
$f_c(Running, t)$,
$f_d(a, b, c, t)$.

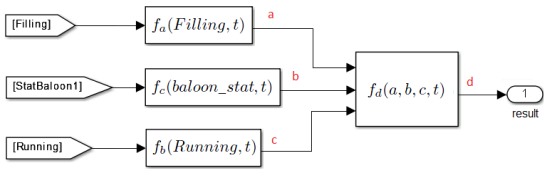


Figure 4.16 Internal functions in property 2

The function $f_d(a, b, c, t)$ is constructed by using an *Embedded Matlab* function and its output is updated at each time step t after the preceding functions a,b and c. The output of this function over the time is represented in Equation 4.5. Moreover, whenever the *Running* signal is true and when the *Filling* signal is true, it must be remain true until the *BaloonStat1* signal becomes true (Full = 1). The code corresponding to MATLAB Function 2 is provided in Appendix B.

Formalization: Let $\mathbb{M}$ be a Simulink model, and property 2 denoted by p:

$\mathbb{M} \models p \quad$ iff $\quad G\ f_d(a, b, c, t) = true$

Proof: The block function of this property can be written as: $f_d(a, b, c, t)$, where the output should hold *true* during the execution at any time step t. This property has three inputs and one output which is connected to a *Proof Objective* block (P-block).

$$f_d(a,b,c,t) = \begin{cases} 0 \quad \text{if} \quad (\neg c_{(t)} \wedge \neg Pre(b,t) \wedge \neg b_{(t)}) \\ \qquad\quad \vee\ (c_{(t)} \wedge Pre(a,t) \wedge \neg a_{(t)} \wedge \neg b_{(t)}) \\ 1 \qquad otherwise \end{cases} \tag{4.5}$$

Property 3

The goal of this property is to assure the model does not generate any false alarms. To address this property we have designed a statement that validates the opposite condition and applied a *'not'* to it.

This property ensures that the controller never ends up in a state where we have *'Filling == true'* (which means the process is undergoing) and we have the alarm light on without having detected any anomaly. The anomaly detection always sets the variable *'bError'* to *true*. Hence, if *'bError'* is set to *false*, means there is no anomaly and the state 'Alarm.Off' in the controller is active.

Figure 4.17, illustrates the implementation of this property. The input *Filling* is used to bound the verification to the time that the system is in the balloon inflation process.

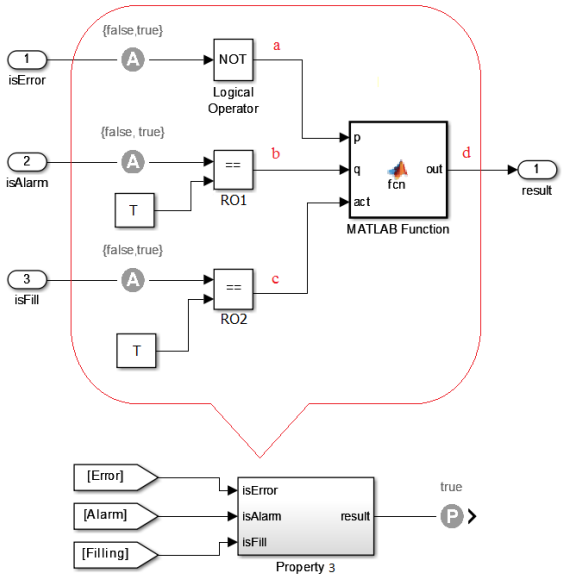


Figure 4.17 Property 3

Formalization: Let $\mathbb{M}$ be our Simulink model, and property 3 is denoted by p:

$\mathbb{M} \models p \quad \text{iff} \quad G\ f_d(a, b, c, t) = true$

Proof: The block function of this property can be written as:
$f_d(a, b, c, t)$, where its output should hold *true* during the execution of the program at each t time steps. This property has three inputs and one output which is connected to a *Proof Objective* block.

As shown in Figure 4.17, the output of *Embedded Matlab* function at each time step, is also the result of the property and should be hold *true* at any time step t. The value of d over time in the Figure 4.17, corresponds to the output of the function $f_d(a, b, c, t)$ which is already declared in Equation 4.5. The function f_d is constructed by using an *Embedded Matlab* function and its output is updated at each time step t after the preceding blocks (AND, RO1 and RO2).

Property 4

The goal of this property is to validate that the model do not permit the pressure within the two balloons to exceed the maximum value configured by the user. This property can be modeled in simulink as represented in Figure 4.18.

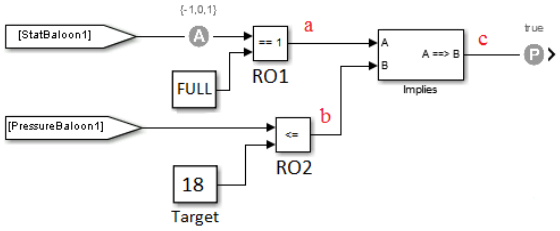


Figure 4.18 Property 4

We have modeled this property by performing a simple comparison between the balloon pressure and the target pressure when the status of balloon is reported as full by the controller.

$$f_c(a, b, t) = \neg a_{(t)} \vee b_{(t)} \tag{4.6}$$

The *Relational Operator* block RO1 validates if the balloon status is full by comparing the input value *BaloonStat1* to the constant FULL ($-1 = empty, 0 = notfull, 1 = full$). In addition, RO2 validates if the pressure reported by the controller at the time step t is equal to the configured target pressure. The output value of the *Imply* block over the time should be always *true* to prove that the property is satisfied. The function corresponding the this block can be represented as Equation 4.6. This property also has two inputs, and its output is updated at each time step t after the preceding blocks (RO1 and RO2).

4.5.2 Verification

Formal verification of specified properties are done on a computer having an Intel Core i7 CPU with 6GB of RAM. We employed Matlab version 2013b and Simulink Design Verifier toolbox is also used as the verification engine.

Table 4.7 depicts the result of the property proving for different properties with their parameters. Since we had two different implementation for the *Function-call Subsystem* block in property I, they are denoted as 1_1 and 1_2 in the result table. (Implementations are shown in Figures 4.13,4.14a).

As can be seen in results illustrated in Table 4.7, the verification time varies in different properties. Some has shorter and some has longer analysis time. The column *Elapsed* in results table, specifies the time that verification engine spends to find appropriate test cases to prove or disprove the corresponding property. The column *Params & Conditions* in the table denotes parameter values and conditions that are set to verify a particular property.

For instance, based on the design of the system, a cylinder has a movement lag which is defined as 2 seconds by default. In addition, each cylinder movement increases the pressure equal to 6 cm of water, and if the target pressure for balloon is defined as 18 cm (Empty balloon has 6 cm), the cylinder needs to move two times. Every request for inflate a balloon should be done within 15 seconds which is defined in *InflateTimer*. As such, if the cylinder lag set to 7 seconds, the total inflation time including time spent for other states exceeds 15 seconds, as a result the property becomes falsified.

In fact, complex properties spend more analysis time than others. On the other hand, using the *Proof Assumption* block and defining the appropriate parameter value , is one of the ways that can reduce the verification time.

4.6 Conclusion

In this paper, we used the formal approach and model-based design in order to specify and formally verify the functionalities of a medical device known as Endotracheal intubation. We proposed a formal model of the system as well as specification blocks of its linear properties, and formally verified the specified properties using Simulink Design Verifier. The system is modeled with parallel components in Simulink/Stateflow, where the event passing/handling and synchronization is efficiently provided. The chart is also optimised to avoid having possible issues such as State Inconsistency, Conflicting transitions, Data Range Violations and Cyclic Behavior in the Stateflow. We also employed Simulink Design Verifier toolset to prove correctness of the model as well as the safety and some temporal properties. In

Table 4.7 Verification results

Property	**Params & Conditions**	**Elapsed**	**Result**
1_1	Fill attempts > 2	$17_m, 42_s$	Falsified
1_1	Fill attempts = 2	$17_m, 43_s$	Satisfied
1_2	Fill attempts > 2	$15_m, 14_s$	Falsified
1_2	Fill attempts = 2	$15_m, 18_s$	Satisfied
2	Cylinder lag=7 secs	$1_h, 3_m, 24_s$	Falsified
2	Cylinder lag=4 secs	$1_h, 3_m, 24_s$	Satisfied
3	Cylinder lag=2 secs	3_s	Falsified
3	Cylinder lag=6 secs	$2_h, 4_m, 25_s$	Satisfied
4	Set point=18, Target <=18	$1_m, 10_s$	Satisfied
4	Set point=18, Target <16	$30_m, 56_s$	Falsified

this effort, property proving as well as simulation traces for different components is done based on the discrete timing concept. In addition to the previous work, the subsystem advantage is used for encapsulation the specified properties. Moreover, the capability of triggered subsystems to handle the event from another components in the model is also used. The authors plan to extend this work for more linear temporal properties, in order to overcome the limitations of the Simulink for specifying such properties. These properties will be available through Simulink library as different customizable blocks.

CHAPTER 5 ARTICLE 3
Using Design Verifier for Proving some LTL Properties

Mohammad-Reza Gholami, Hanifa Boucheneb
International Journal of Intelligent Information and Database Systems (IJIIDS)

Abstract– The increase in sophistication in embedded systems such as in healthcare and avionics, requires design of safety-critical applications to use more systematic processes for development. Traditional design processes are not responsive enough in identifying the flaws in requirements; thus, the whole process would be more expensive and take longer. Matlab/Simulink is an often used industrial tool in designing embedded systems. One of the primary uses of Simulink is modeling the embedded software and its physical environment in a common formalism. This feature of the tool renders it highly valuable in the validation of embedded software design, leveraging numerical simulation. Having claimed this, formal verification of such models still proves problematic, as Simulink is a programming language without enough documented formal semantics. In this paper, we propose a technique to facilitate formalizing some LTL properties so that they can be added to the Simulink model as some customized blocks. We also present how the Simulink model can be instrumented by using proposed custom library and show the way that the functionality is formalized.

Keywords– Formal Methods; Formal Verification; Design Verification; Model-Based Design; Linear Temporal Logic; Safety-Critical.

5.1 Introduction

In most day-to-day activities that we undertake, software performs a crucial role. Such diverse activities as driving cars, operation aircraft navigation systems, and working in an office all require software. In the last few years, reliability of complex hardware and software systems has become progressively vital. As a result, the need for verification of software is a major problem in the world of computer science. The systems used today are real-time which require more effort to find defects, and as a result, validation and verification activities are becoming quite costly. The essence of software validation cannot be over-emphasized given that some software can cause catastrophic repercussions to human life. When the catastrophic consequences are caused by failure in a computer system, the system is termed as *safety-critical*.

In software engineering a crucial goal is to come up with a system that operates reliably irrespective of its complexity. That is to say, employing formal methods which are mathematically based languages, techniques and tools, is a prospective approach to attaining this goal. With regard to a safety critical system, having one single error in the source code may causes an unexpected behavior of the system. Consequently, it can even incur high costs and could possibly compromise people's well being. With regard to critical real-time embedded systems, validation and verification activities are becoming quite costly with the growth in the size and complexity of these systems.

Applying a Model-Based Design [33] for safety-related and large applications, and employing formal verification techniques, demonstrate that a system or its software component satisfies its accurate criteria. For conducting a formal verification, the first step is to define the formal model of the system. Additionally, accessibility of executable models to perform validation, verification and test is the most operative factor of Model-Based Design that aids in applying formal verification techniques.

This paper is organized as follows: Section 5.2 presents some background and previous work connected to this research topic. Description of the employed tool and its components along with their semantics are placed in Section 5.4. We depict implementation of our properties in Section 5.5. The results of our implementation are evaluated in Section 5.6. Finally, Section 5.7 forms the conclusion.

5.2 Background And Related Work

The abstract representation of a system is what forms a model. A software model refers to the way of expressing a software design, so as to depict the design, some sought of abstract figures or language typically used. When conducting software modeling, there is a need to include interactions with other software, interfaces and the entire software methods [34]. A modeling language can be used by engineers to model the system which can be textual or graphical [39].

5.2.1 Model-Based Design

The identification of errors is done late by traditional design processes and hence, the entire process becomes more costly and long. As for the model based design, the complexities and difficulties are addressed by the provision of an executable specification which denotes that the model is more than just a document. It is a crucial part of the design process [54]. It further offers a single design environment that enables developers to utilize a single

model of the whole system for purposes of model visualization, single design, data analysis, testing and validation and finally the deployment of the product, with or without automatic code generation [46]. Additionally, model-based design establishes a structure for reusing the software that permits established designs to be reliably and effectively upgraded in a cost effective and more simplistic way. This sort of design concentrates on utilizing executable system models as the basis for the, implementation, specification, test and design verification [13].

Simulink® can be used to build an executable model. This is an environment for simulation that is multi-domain and model-based design for embedded and dynamic systems. It offers a graphical environment that is interactive and a customizable set of block libraries that help the user to simulate, design, implement and test a wide range of time-varying systems that include controls, signal processing, image processing and video processing.

5.2.2 Formal Methods

These methods improve the process of verification by using formal concepts and notations in writing specifications and requirements. These methods offer different degrees of formal proof that is used within the process of verification. It can also be used to augment the validation process, by allowing the properties and consequences of non-executable specifications to be identified through a theorem proving at the early stages of the life cycle. Also in the formal methods, logical and mathematical techniques are used to investigate, express and analyze the specification, documentation, design and behavior of both software and hardware.

The term *formal* in formal methods comes from *formal logic* and implies "to do with form" [66]. In formal logic, reliance on judgment and human intuition is evaded in assessing the arguments. In order to constitute a valid proof or an acceptable statement, formal logic utilizes a constrained language with precise rules for writing theorems, assumptions and proofs.

5.2.3 Formal Verification

Formal verification in respect to software refers to the automated proof of properties that are specified on the code without the program being executed. Further, it ascertains that a design is aligned to some notion of functional correctness which are expressed precisely [18]. The main benefits of formal verification relative to testing or dynamic verification, are its exhaustiveness and soundness. Specifications in formal methods are the *well-formed* statements that are mathematical in nature and are used to specify a property that is to be

verified in the system [10].

In formal verification, the mathematical representation of the system refers as a model and model checking [24] is one of the most common formal verification techniques. By applying model checking technique and having design's model along with a temporal logic formula used to describe a specification, can determine whether or not the model satisfies the specification [44].

5.2.4 Related Work

Simulink is a platform for model-based Design of embedded systems from Mathworks. Simulink Design Verifier is a toolset that uses formal analysis for property proving. Although Simulink design Verifier provides some temporal operator blocks in its library, but specifying LTL properties is not potentially straight forward. For this reason, proving linear temporal properties in a Simulink model is done by transforming the model into input language of another model checkers.

Recently, efforts have been exerted in transforming the Simulink model. In particular, for an embedded system application Yelamuri [64] presented a new technique to generate *C-code* from Simulink models. In the proposed technique, FLEX and BISON are used as code-generating tools as well as generating the model parser which can then extract detailed information pertain to blocks and lines in the model and create tree of blocks. The primary motivation in [53] presenting a translator algorithm along with a tool that can automatically translate a subset of Simulink model into NuSMV model checker as an input language. Meenkashi et al. believe that using the proposed tool shortens the process of formal verification of safety avioncs components with less error.

Christian Heinzemann et al. in [40], proposed a model-2-text transformation that is used to transform instances of the given Simulink EMF-Model. Similarly, Pajic et al. In [58], presented a matlab plug-in tool known as UPP2SF that can be used for translation of models from UPPAAL to Simulink/Stateflow. The proposed tool also enables UPPAAL models to be simulated and tested in Simulink/Stateflow.

For the verification of properties, there are works describing the translation of Simulink models into various model checking languages, including NuSMV, Lustre, SAL and Promela/SPIN [55, 53, 72, 59, 50].

With attention to linear temporal properties, efforts have been exerted recently in describing LTL verification of Simulink models. As an illustration, Miller et al. in [55], propose using the symbolic model checker NuSMV [23] as verification tool. Since the simulink model can

not be used directly as an input for NuSMV model checker, as a result it should be first being translated into synchronous dataflow language Lustre [36] and then into NuSMV. Similarly, the primary motivation in [12] was to verify the correctness of Simulink models with respect to a set of specifications given as LTL formulae. The authors applied the explicit model checking technique, after initially formalising the simulink models based on the set-based reduction concept. it is done to reduce the state space and support for non-determinism of input. In terms of operational semantics, formalization of Mathwork's Stateflow is described in [37] by Hamon et al. Similarly, Bouissou [19] also outlined the operational semantics of Simulink engine.

Above mentioned works were done on transforming between Simulink model into another model checker tools. In our work, we propose formal verification of Simulink models based on extending the Simulink library with some customizable blocks. In addition, the proposed blocks have support for some LTL properties and facilitate the process of instrumenting the models with properties. Moreover, The generated model can also be analysed by Simulink Design Verifier.

5.3 Temporal Logic

The other type of logic, symbolic logic, supports generally the reasoning with propositions, i.e., with statements to be evaluated to false or true. Temporal logic is a branch of the broad symbolic logic concentrating on propositions whose truth values rely on time. That is contrary to the classical logic point of view where the truth value of a recurrently spoken proposition must always be the similar and must neither depend on the mechanisms of additional information nor on the repetition. Temporal propositions classically contain some (implicit or explicit) reference to time conditions, while classical logic is involved in timeless propositions [47].

5.3.1 Linear Temporal Logic

LTL (*linear temporal logic*) formulas are constructed from predicates through the usual propositional connectives ($\vee, \wedge, \Rightarrow, \neg$) and two temporal operators: $\bigcirc$ and $\cup$ [70].

Following this logic, one illustrates the distinct properties of different paths making up a computation tree. Specifically, following this logic can express such properties as 'for every n consecutive states' or 'for some state on the path'. This logic is known as a linear temporal logic, or simply LTL.

Formulas such as these are satisfied through computations. These are functions which assign

truth values to the proposition elements at each time step [31].

Semantics of LTL formulas [45] are defined as follows:

— If A is a formula, then $\neg A$ is a formula.
— If A is a formula, then $\bigcirc A$, $\diamondsuit A$ and $\square A$ are formulas.
— If A and B are formulas, then $A \cup B$ is a formula.

The symbols $\bigcirc, \diamondsuit, \square, \cup$ are called *temporal operators*

In advance of formally defining the semantics of LTL formulas, we must first try to explain their meaning more informally. LTL formulas are statements - either true or false - of computation paths, or sequences of states $s_0, s_1, s_2, ...$

Semantics of LTL formulas is defined as follows:

Let $\pi = s_0, s_1, s_2, ...$ be a sequence of states and A be an LTL formula. We define the notation A *is true on* π, denoted by $\pi \models A$, by induction on A as follows. For all $i = 0, 1, ...$ denote by π_i the sequence of states $s_i, s_{i+1}, s_{i+2}, ...$ (Note that $\pi_0 = \pi$).

— $\pi \vDash \neg A$ if $\pi \nvDash A$.
— $\pi \vDash \bigcirc A$ if $\pi_1 \vDash A$;
— $\pi \vDash \diamondsuit A$ if for some $i = 0, 1, ...$ we have $\pi_i \vDash A$;
— $\pi \vDash \square A$ if for all $i = 0, 1, ...$ we have $\pi_i \vDash A$
— $\pi \vDash A \cup B$ if for some $k \geq 0$ we have $\pi_k \vDash B$ and $\pi_0 \vDash A, ..., \pi_{k-1} \vDash A$.

When we consider a path π and paths π_i as in this definition, instead of saying that a temporal formula A is true on π_i we will sometimes say that A is true at the state s_i on the path π.

Semantics of Temporal Operators are illustrated in Figure 5.1, and can be explained as follows:

$\bigcirc$ (next) The formula $\bigcirc A$ holds, if A holds at the next state on the path.

$\diamondsuit$ (eventually) The formula $\diamondsuit A$ holds, if A eventually occurs, i.e., A holds at some state on the path.

$\square$ (always) The formula $\square A$ holds, if A holds globally, i.e., at every state along the path.

$\cup$ (until) The formula $A \cup B$ holds, if A holds until B occurs, i.e., there is a state on the path at which B holds, and at every state before A holds.

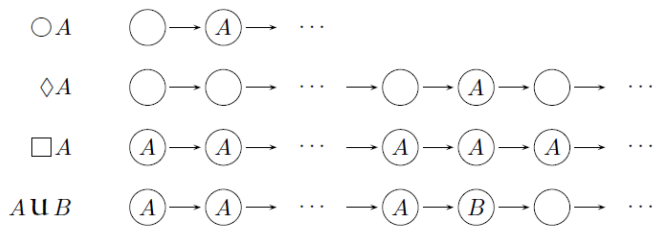


Figure 5.1 Semantics of Temporal Operators

5.4 Simulink and Stateflow

By definition, Simulink is a platform for model-based Design and multi-domain simulation of dynamic systems. Stateflow, on the other hand, is a model-based development environment that is widespread and is used in several industries, such as medical, aerospace and automotive. Particularly, Stateflow diagram facilitates the graphical representation of parallel and hierarchical states together with transitions between them and inherits all code and simulation generation capabilities from Matlab toolset. Following this section, Simulink and Stateflow semantics are briefly described.

5.4.1 Simulink

Simulink helps the design and simulation of wide range of systems by providing an interactive environment along with collections of customizable blocks. It includes extensive library and toolboxes of functions commonly employed in modelling a system.

Simulink Library

Simulink comes with a standard block library whose blocks are placed in different categories. In addition, we can create our own block libraries as a way to reuse the functionality of blocks or subsystems in one or more models. The basic work flow for creating and using our custom block libraries (*VeriForm*) with Simulink blocks is presented in the implementation section.

In the following, some of the most common Simulink blocks from the standard library is briefly described [6]. These blocks are used to specify some LTL properties.

Inport blocks connect outside of a modeled system into the system, and can be found in *Ports & Subsystems, Sources* library. Simulink® [6] software assigns Inport block port numbers automatically within a top-level system or subsystem sequentially, starting with 1. *Outport* blocks are connections from a modeled system to a destination outside of the system, and can

be found in *Ports & Subsystems, Sinks* library. Simulink® software assigns Outport block port numbers automatically within a top-level system or subsystem sequentially, starting with 1. The *Constant* block produces a constant value with the type real or complex. The *Relational Operator* block performs a relational operation on its two inputs and produces output. Given operators can be equal, not-equal, smaller than, smaller or equal, greater than, and greater than or equal. The *Logical Operator* block is being used to perform the specified logical operation on its given inputs. The supported operations consist of AND, OR, NAND, NOR, XOR, NXOR and NOT. The *Sum* block applies addition or subtraction on its given inputs. It has no state and the sample time for this block is also inherited from driving blocks. The *Unit Delay* block holds and delays its given input by the sample period specified as parameter. In other words, *unit-delay* block gives the opportunity to change the sample time of the signal.

Figure 5.2, denotes how the above mentioned blocks are displayed in the Simulink standard library.

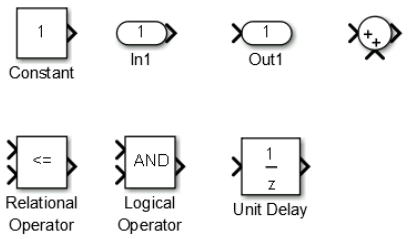


Figure 5.2 Simulink sample blocks

Simulink Block Methods

Blocks represent multiple equations which are represented through *Output* and *Update* types known as block methods. By running a block diagram these methods can be evaluated too.

A simulation loop is used to evaluate block methods in which each cycle through the simulation illustrates a block diagram evaluation at a specific point in time. As such, at the current time step, outputs of each block as well as its states at the previous time step, are calculated by the output method depending on the block inputs. Likewise, discrete state of each block at current and the previous time step are calculated by update method.

A Simulink block has sets of *input* and *output* ports. A block with N input and M output ports defines a function which describes each of the signals at the output ports as a (possibly time-dependent) expression of the signals at the input ports. Formally, a block is a tuple

(P_i, P_o, f), where P_i is the set of input ports, P_o is the set of output ports and $f : \mathbb{R}^{\mathbb{N}} \rightarrow \mathbb{R}^{\mathbb{M}}$ is a function which defines the behavior of the block [52, 22].

Some Simulink blocks [52, 37, 21] with their ports and functions are denoted in Figure 5.3.

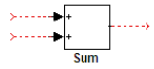
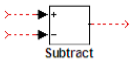
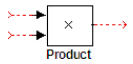
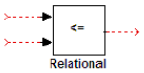
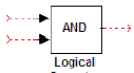
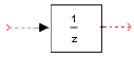
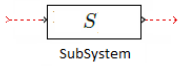
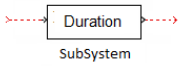
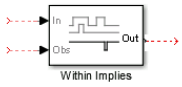
Block	Ports	Function
	$P_i = \phi$ $P_o = \{m_1\}$	$m_1(t) = K$
	$P_i = \{n_1, n_2\}$ $P_o = \{m_1\}$	$m_1(t) = n_1(t) + n_2(t)$
	$P_i = \{n_1, n_2\}$ $P_o = \{m_1\}$	$m_1(t) = n_1(t) - n_2(t)$
	$P_i = \{n_1, n_2\}$ $P_o = \{m_1\}$	$m_1(t) = n_1(t) \times n_2(t)$
	$P_i = \{n_1, n_2\}$ $P_o = \{m_1\}$	$m_1(t) =$ true, iff $n_1(t) \leq n_2(t)$
	$P_i = \{n_1, n_2\}$ $P_o = \{m_1\}$	$m_1(t) =$ true, iff $n_1(t) =$ true$\wedge n_2(t) =$ true
	$P_i = \{n_1\ \}$ $P_o = \{m_1\}$	$m_1(t) = n_1(t\ \ + 1)$
	$P_i = \{n_1\}$ $P_o = \{m_1\}$	$m_1(t) = f(n_1)$ with $f = \mathcal{E}\{S\}$
	$P_i = \{n_1\}$ $P_o = \{m_1\}$	$m_1(t) =$ true, iff $0 \leq t \leq n_1$ Otherwise $m_1(t) =$ false
	$P_i = \{n_1, n_2\}$ $P_o = \{m_1\}$	Assume t in {0,...a,....b,....} m1(t+1) = false iff n1(t_a U t_b) = true and n2(t_a ... t_b) <> true for at least 1 step Otherwise m1(t) = true

Figure 5.3 Simulink blocks with ports and functions at time $_t$

As an illustration, the Sum block in Figure 5.3 has two inputs and one output ports. In addition, output of the block at time step t, equals to the addition of values of its input ports at time t, and the block function is shown as $m_1(t) = n_1(t) + n_2(t)$.

Simulink Semantics

Simulink has a plethora of semantics (depending on options that are configured by the user), which are informally and often times only partially documented.

Regarding Simulink timing, it is a known fact that the discrete-time Simulink signals are piecewise-constant, continuous-time signals [72]. Associated timing information can be linked to these signals, referred to as *sample time*. Furthermore, the sample time of a signal shows exactly when the signal is updated in the model. When the sample time equals zero, the block is identified as having continuous sample time. This means that, it executes at every point in time. When sample time has a value greater than zero, the block is identified to have discrete sample time.

In Simulink, a discrete block executes at sample time points, and remains constant in the intervals between these sample time points. In like manner, Simulink block methods such as *Output* and *Update* methods are executed at each sample time.

Simulink Block Priorities

Update priorities to blocks can be assigned explicitly. The output methods of each block in the model are executed depending on the their priorities from higher to lower priority. If there is consistency with block sorting rules the priorities can be honoured. Moreover, if the execution order of the block is set explicitly by setting block priorities within a subsystem, Simulink removes those block priority settings when the subsystem is expanded. Simulink checks the block properties in the following order:

— Sample time (faster rate first)
— Priority (lower priority number first)
— Port number (lower input port number first)

5.4.2 Stateflow

Simulink is used to model the continuous dynamics and Stateflow is used to specify the discrete control logic and the modal behavior of the system [71]. The Stateflow modeling language is based on hierarchical state machines with discrete transitions between states. It employs a variant of the finite state notation of machine as establsihed by Harel [38] and offers the elements of language needed to describe complex logic in a readable, natural and understandable form. Given that it is strongly integrated with Simulink and MATLAB, it can offer an environment efficient enough for designing embedded systems that contain supervisory and control.

A state is referred to as *superstate* if there are other states in it and a *substate* when it is held in another state. When a state comprises of two or more substates, it has *decomposition* that can be either parallel (AND) or exclusive (OR) decomposition. All substates at a given level in the hierarchy of the Stateflow must have the same decomposition. In parallel (AND) decomposition, states can be active simultaneously and the activity of each parallel state is independent of all other states.

Defined *Events* can be used to trigger actions in parallel states of a Stateflow chart. One way of triggering an action and/or transition is through broadcasting of an event. The execution of *Actions* can be either as part of a transition from one state to another or based on the activity status of a state which can be ***du**ring* ***ex**it*, ***en**try* and ***on** event* actions.

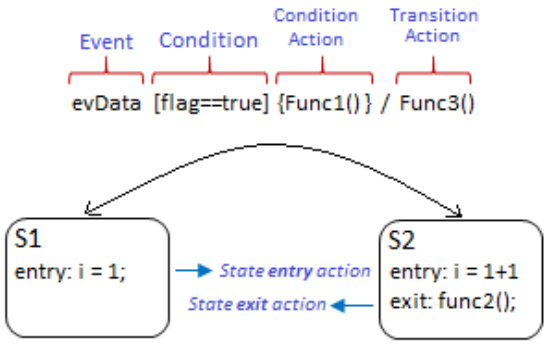


Figure 5.4 Stateflow Semantics

A transition in a Stateflow is represented as an arrow between two states in Figure 5.4. It depicts the behavior of a condition, simple event and transition action detailed on a transition from one exclusive (OR) state to another. At first, state *S1* is active and *Entry* action is then executed. Upon receipt of the event *evData* the chart root identifies a valid transition to state *S2* due to the event *evData*, so it validates the condition and if the result is true, the *Condition Action* gets immediately executed and completed. The state *S1* is marked as inactive and the *Transition Action* is executed and completed just when the transition destination *S2* has been established to be valid. In other word, whenever a transition is enabled (Source state is active and condition is true), its corresponding condition action takes place before the transition action is taken. In contrast, transition actions occur only after the source state of a valid transition becomes inactive, i.e., the transition is taken.

For instance, transition from *S1* to *S2* can occur when *S1* is active, event *evData* occurs and the condition *flag==true* is also valid. States can have diverse actions such as: *entry*, *during*, *exit*, and *on event-name* which are being executed based on the current status of the active state. In like manner, *(i=1)* is executed whenever the state Start becomes active and entry

action is executed. Moreover, *func2()* is called whenever the state *S2* loses control and its exit action is executed.

5.4.3 Simulink Design Verifier

Simulink Design Verifier® [5] (SLDV) is a tool set of Matlab which uses *formal methods* to identify hard to find *design errors* in the models without requiring extensive tests or simulation runs. It uses the Prover Plug-In® [2] *formal analysis* engine, in order to prove the properties. It is known from the documentation of the SLDV, that the Prover Plug-In is based on Stålmarck's proof procedure which was patented in 1992 [68]. In addition, performing bounded and unbounded model checking, sequential and combinational equivalence checking as well as the test generation are some features provided by this tool. Moreover, modelling of sequential systems employing imperative and declarative formalisms is also supported by prover engines. It also supports a wide range of data types including integers, reals, arrays and booleans.

In the following, *Implies* and *Within Implies* blocks from Simulink Design Verifier library which are used in specifying LTL properties, are briefly described:

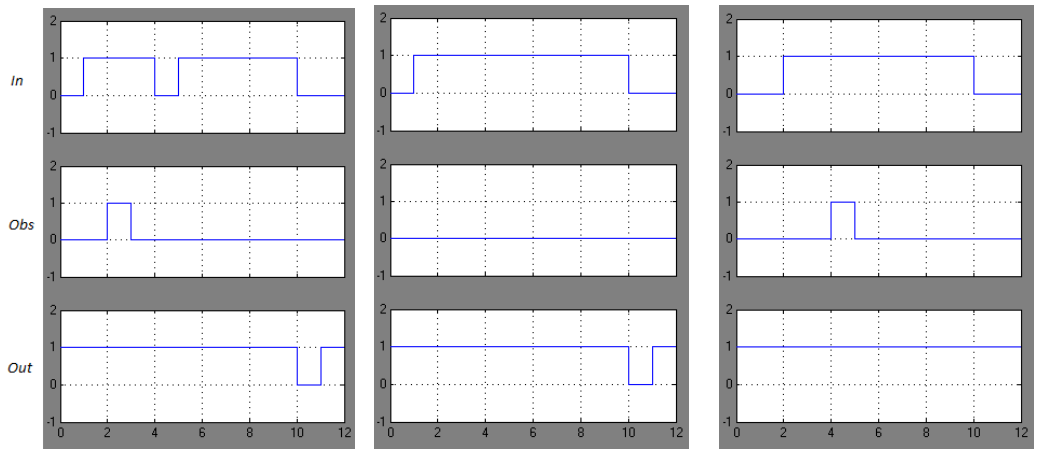
Implies block

The Implies block from Simulink Design Verifier library let the designer to specify a condition to produce a given response. It tests whether the first input implies the second. For instance, if input A is true and input B is false, the output is false; for all other pairs of inputs, the output is true.

Within Implies block

The Within Implies block captures the within implication by observing whether the *'Obs'* input is *true* for at least one step within each true duration of the first input *In*. Whenever *Obs* is not detected within a particular input true duration, the output becomes false for one time step in the step that follows the input true duration. This block captures the behaviour: ('Within' In) $\Rightarrow$ Obs.

To put it another way, the true duration of a signal corresponds to consecutive time steps during which a signal is *true*. Based on the definition of the *Within Implies* block, if *Obs* is not observed within the true duration of *In*, the *Out* becomes *false* for one time step. When there is no true duration of *In*, *Out* remains *true*, and if *Obs* occurs multiple times during the true duration of input, it does not affect the output.



(a) False Output (b) False Output (c) True Output

Figure 5.5 Within Implies Simulink Block

In the example illustrated in Figure 5.5, model sample time is considered as 1 second.

— In Figure 5.5a, although *Obs* is observed within the first true duration of *In* (steps 1...4), but it is not observed within the second true duration of *In* (steps 5...10), so *Out* becomes false for one time step after the *In* signal becomes false (at step 11).

— In Figure 5.5b, *Obs* is not observed within the first true duration of *In*, so *Out* becomes false for one time step after the *In* signal becomes false.

— In Figure 5.5c, *Obs* is observed within the true duration of *In* (at step 4), so *Out* remains true until the end of the simulation.

5.5 Implementation

In Simulink Design Verifier, the generic *Proof Objective* block with the *Implies* block will be used for almost all requirement types except for undesired behavior where an alternative modeling strategy closer to the written formulation were used [51]. In addition, requirement specification is needed to be formalized by using different Simulink blocks. In fact, this procedure is crucial and time consuming when those requirements are either complex or frequently used.

Following this section, the Simulink file format and our proposed tool is described. Some customizable LTL properties which are included in the *VeriForm* library are detailed right after. Since any modification in the Simulink model applies some modifications in the model file, the impact of using different types of proposed properties (e.g. Model Reference, Verification Subsystem) from *VeriForm* library are also discussed.

There are two different ways that the user can employ *VeriForm* library: 1) Using this predefined library when creating the Simulink model, 2) Using LTL2SL tool that can add the chosen block to the previously made Simulink model. In our description, we assume that the provided Simulink model is compatible with Simulink Design Verifier (Some blocks such as continuous-time blocks are not compatible with SLDV).

Using VeriForm library to specify properties is not easy for less-experts, so to help designers we propose LTL2SL tool that facilitates the process. In addition, the user can use LTL2SL tool to instrument the model by choosing any predefined LTL block from the *VeriForm* library and specify which signal should be connected to which port. following this step the user can also specify the value for an input port of the block if needed.

The work flow of the proposed solution is illustrated in Figure 5.6.

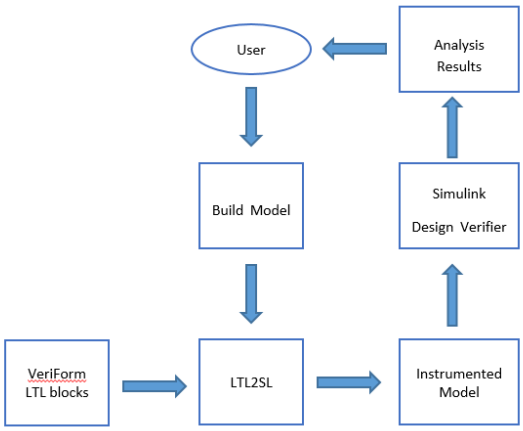


Figure 5.6 Life cycle of the entire process

5.5.1 Understanding Simulink Model

Simulink models are stored as '*.mdl*' files, which contains textual description of the model (properties of blocks and their interconnections along with information required for simulation and graphical display of model) [51],[64]. Hierarchy of model properties is shown in Figure 5.8. Model file contains many properties and only few which are of interest with respect to a code generator are highlighted in the Figure 5.8.

Model file begins with keyword *Model* then enclosed inside curly braces are the model properties, default block properties used in this model and system details.

Additionally, the file starts by defining an outer section called Model. Inside the section, the name of the model is defined by the parameter with the key Name and the value "Minimal".

As a convention, this name should be equal to the actual file name. The next six nested sections *Array*, *Simulink.ConfigSet*, *BlockDefaults*, *AnnotationDefaults*, *LineDefaults*, *BlockParameterDefaults*, contain some configuration parameters and default values, which are not used in the code generation, but have to be present in the file. Sample Simulink model is denoted in Figure 5.7.

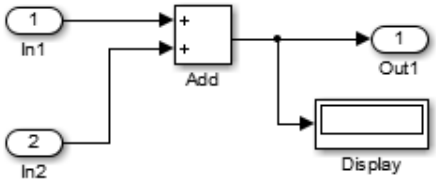


Figure 5.7 Sample Simulink Model

The actual content of the model resides in the *System* section. In Simulink, the basic elements that a model consists of are called blocks. The behavior of the whole model is defined by the individual blocks as well as the connection between them.

System begins with keyword *System* and contains block details followed by line details inside curly braces. Subsystem and Stateflow are treated as a *System* inside another system. Lines normally contain source and destination block names and port numbers. Branch information is included in line details if output of one block is connected as input to more than one block.

Typical blocks are for example constant blocks, subsystems, clocks, or scopes. An important block in Simulink is the *Subsystem* block. A subsystem is used to group blocks and to create a hierarchy in the model. In addition, the Subsystem block contains a specific parameter *Ports* which lists the number of ports for this subsystem. Moreover, a subsystem contains a nested section called System. This sections contains the common *Name* parameter, which needs to be equal to the name of the outer block, and a specific parameter *Open*, which tells the Simulink user interface whether it should display a window containing the contents of that subsystem or not. At last, the *System* section contains all block which are contained in the subsystem. In the example illustrated in Figure 5.7, the model contains two incoming ports, one single outgoing port and *Add* block. Hierarchy of model properties for this example is shown in Figure 5.8.

5.5.2 Property Specifying Process

The steps for creating and using the proposed custom library is described in this section. To begin, we can enumerate the steps as follows:

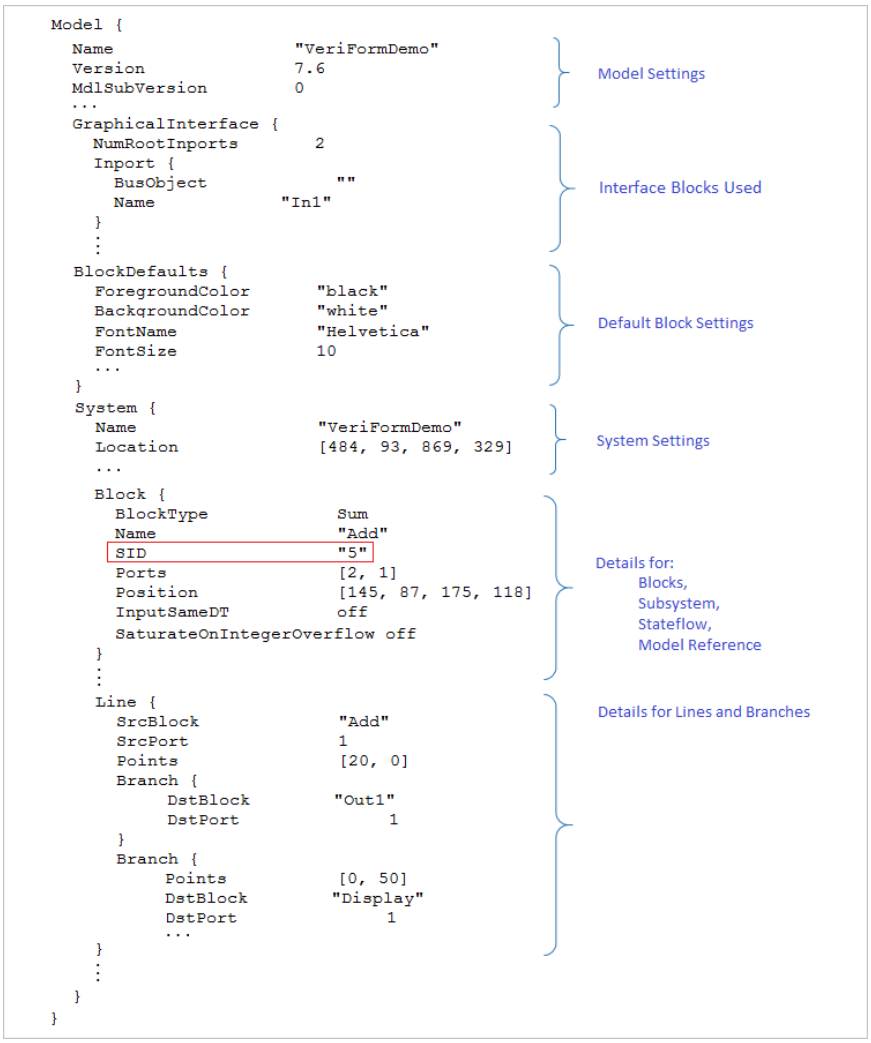


Figure 5.8 Simulink Model File Format

1. Specify the property by using blocks from the standard Simulink library including other subsystems or embedded matlab functions (EMF).
2. Group blocks in the defined property and convert them as a new Subsystem.
3. Create the *VeriForm* Simulink custom library from the resulted Subsystem.
4. Make a parametrized template from the resulted Subsystem to be used by the stand-alone tool.
5. Make a Referenced Model from the resulted Subsystem then make a parametrized template as well to be used by the stand-alone tool.
6. Instrument the model by using the proposed tool or choosing the property from the Simulink library browser where the *VeriForm* custom library is added.

As shown in Figure 5.8, each block has an identifier named *SID* which is unique in each model. Making parametrized template means that for each property a single block will be created and the required items and values are parametrized to simply being used by the proposed stand-alone tool. Figure 5.9b, illustrates the created subsystem from property defined in Figure 5.9a.

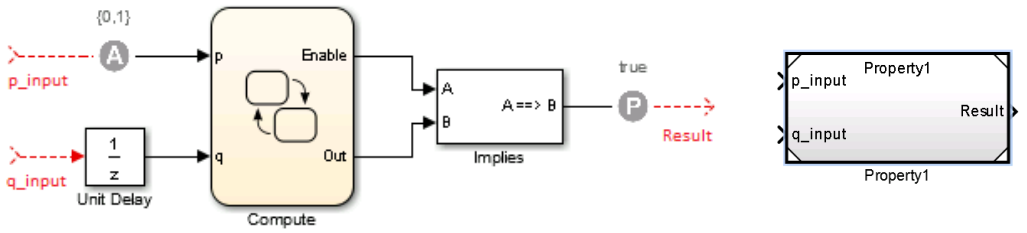
When using the default subsystem block from Simulink library in the instrumentation process of the model, the definition of all the blocks which are grouped and encapsulated in the Subsystem will be transferred to the generated model. By using *Model Reference* instead of default *Subsystem* block, we can make the process lighter while we also made the logic of the property hidden. The proposed method suggests using model reference which is also configured as an atomic. The difference between generated model code for the *Subsystem* block and *Model Reference* is illustrated in Figure 5.10.

The implementation of this method makes the generated model more simple and easy to debug. For instance as per shown in Figure 5.10b, just a tiny code corresponding to inputs and outputs of the property is added to the Simulink model file.

5.5.3 Definition of Properties

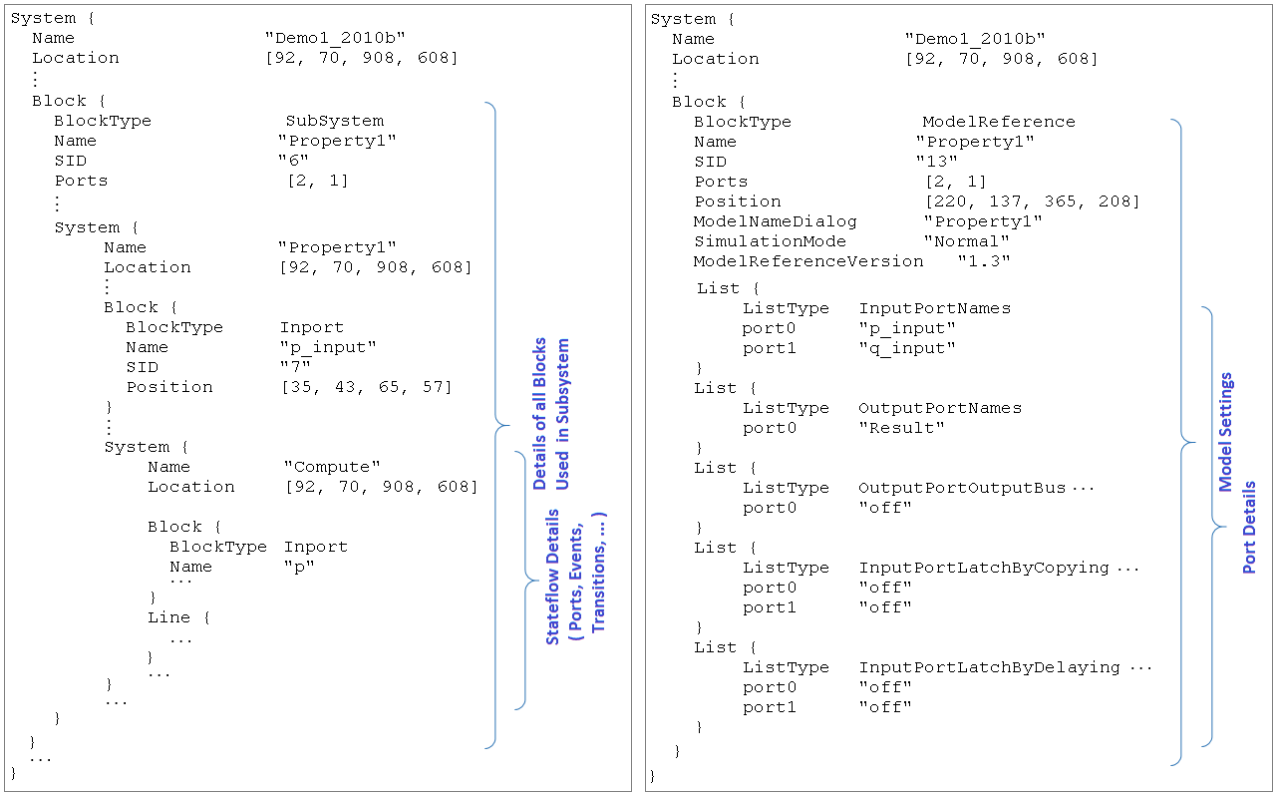
Following this section, the proposed custom properties will be described along with their inputs and outputs. We assume that $\mathbb{M}$ is the provided Simulink model, and selected outputs of blocks inside $\mathbb{M}$ are connected to inputs of proposed properties.

The term $Pre(x,t)$ is used for *Predecessor* in provided equations when defining specified properties (It corresponds to *Unit Delay* block in Simulink library). As such, The predecessor of signal x at time t is denoted by $Pre(x,t)$ that corresponds to the value of that signal at time step $t-1$. The term $Pre^*(x,t)$ is also used as the transitive closure of $Pre(x,t)$.



(a) Part of a sample model (b) Model Reference

Figure 5.9 Converting group of blocks to a Model Reference



(a) Subsystem Code (b) Model Reference Code

Figure 5.10 The difference between *Subsystem* and *Model Reference*

Property Eventually p

This LTL property denoted by $\Diamond p$, holds for a given path π, if p eventually occurs in some state of π. We can not directly implement this property with temporal operators provided by Simulink Design Verifier, because it only allows verification of assertion-based temporal properties and the verifier doesn't know when the execution path terminates. To prove this property, we can use the negation so the violation of the property corresponds to the formula $\neg(\Box\neg p)$. As such the verifier checks if there is any state in the path π that doesn't hold $\neg p$.

Let p be *desired_val* = *signal*. The proposed block concludes the desired value as the second input to apply it in the condition. The implementation of this property is illustrated in Figure 5.11.

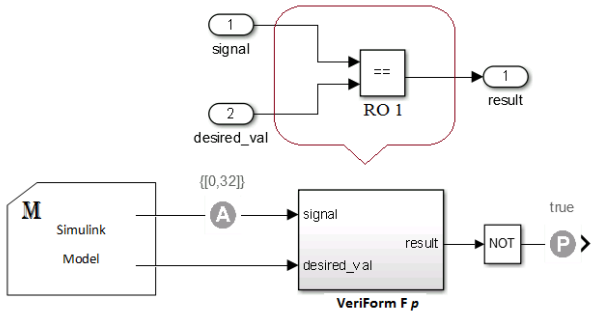


Figure 5.11 LTL F p

This property has two inputs, one for the signal corresponding the current value of *signal*, and the second one as mentioned earlier is used for the condition (at each time step, p is satisfied iff signal = desired_val). For instance if we want to verify the signal eventually has the *desired_val* of 5, the condition is made based on the *signal*'s value.

The sample time for the entire model is set to 1 second. Since this part is added lately to the model so the execution order of the Relational Operator block *RO1* is lower than the other blocks in the model(every block added to the model has a sorted order starting from 0, and blocks with lower sorted order number have higher execution order). The evaluation of the *Relatinal Operator* block diagram which is denoted in Figure 5.3 is performed at each time step. Its output is also available to the *Proof Objective* block (P-block) at the same time step which specified to have execution order than *Relational Operator* block.

The *Proof Objective* is used in this custom block and validates the output for *false*. If the result of the design verifier shows that the property is not satisfied, in fact it found that in a particular step of the execution the signal held the desired value and $(\Box\neg p)$ is not true, consequently the property $\Diamond p$ is satisfied.

Property Eventually p in $_{[0,n]}$ steps

This LTL property denoted by $\diamondsuit_{[0,n]}p$, holds for a given path π, if p eventually occurs within the first n steps of π. The desired specification for 5 time steps can be modelled in LTL as:

$$\varphi = \diamond_{[0,5]}p$$

where $\diamond_{[0,5]}p$ is an abbreviation for 'p may be satisfied within five steps of time' and is defined by:

$$\diamond_{[0,5]}p = p \vee \circ p \vee \circ \circ p \vee \circ \circ \circ p \vee \circ \circ \circ \circ p \vee \circ \circ \circ \circ \circ p$$

Satisfaction of formula φ requires that at least at one time step p holds, that is, communication will always occur within the next 5 time units.

To implement this property we used the standard Simulink library to build a required amount of time steps and a temporal operator *Within Implies* which is provided by Simulink Design Verifier library. As per definition of *Within Implies* block, it captures the within implication by observing whether the 'Observer' input is *true* for at least one step within each true duration of the first input In. By employing of *Unit Delay*, *Sum* and *Relational Operator* blocks we first constructed a limited steps of execution, and then, the output of the first part is fed to *Within Imply* block. A *Proof Objective* with the desired value of true is used to let the design verifier prove if the property is satisfied or not. The implementation of this property along with the execution order of the blocks are illustrated in Figure 5.12.

At each time step, the Relational Operator block *RO1* evaluates if the property did not exceed the specified time steps, and ensures that the input of the *Whithin Implies* block becomes *false* as soon as the desired duration is passed. The *Relational Operator* validates the input signal against the desired value. This signal then becomes the 'Observer' input of the temporal operator *Whithin Implies* block and afterwards the *Within Implies* block checks if the Obs was true for at least one time step during the specified number of time steps.

The purpose of using the *Unit Delay* block is to prevent causing the *Algebraic-loop*. As depicted in Figure 5.3 the evaluation of the *Sum* block is done in each sample time, so direct feedback the output of the Sum block to one of its inputs causes algebraic loop and will be reported as an error in compile time. In other words, when using unit-delay, once the output of all blocks at time t is computed, the internal state of unit-delay block can be computed and this internal state will be the input of the *Sum* block at the next time step (Predecessor of time step $t + 1$).

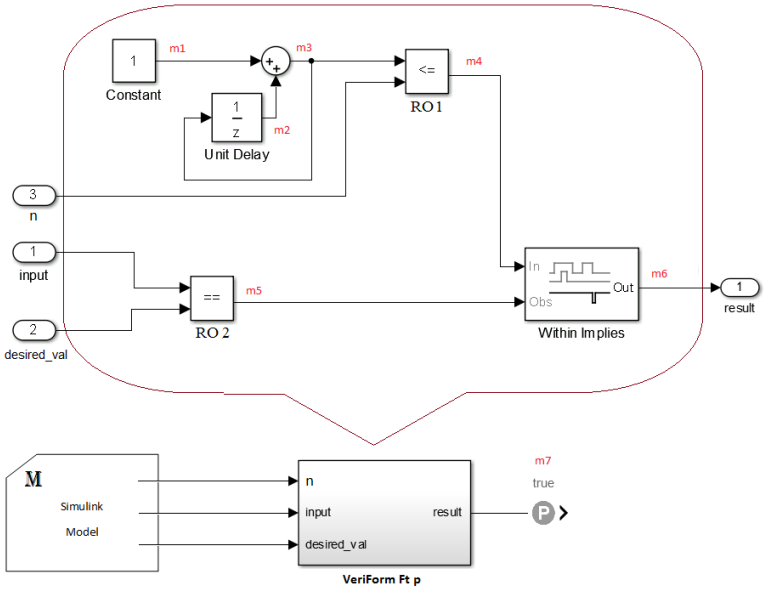


Figure 5.12 LTL $F_{[0,n]}\ p$

This property comes with three inputs, first is the *input* signal, second is the *value* that should be held eventually within n time steps in the path π, and the third defines the *duration* steps n. In particular, specification of the property is located in the block illustrated in Figure 5.12, and its output represents p.

Simulink model $\mathbb{M}$ can have different inputs and outputs, and depend on the chosen property, desired outputs are connected to the property block to get verified.

Moreover, the unit-delay has an initial condition set to zero, so at the first time step, its output will be the value of the initial condition.

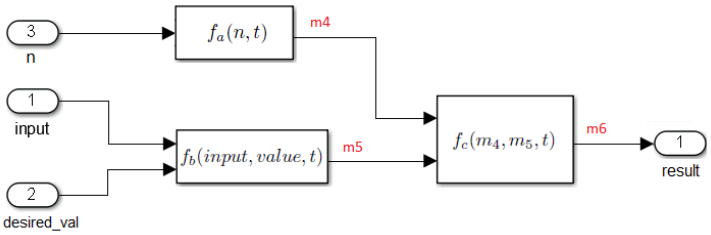


Figure 5.13 Internal functions in $F_{[0,n]}\ p$

Formalization: Let $\mathbb{M}$ be a Simulink model and p be a formulae:

$\mathbb{M} \models \Diamond_{[0,n]}\ p \quad \text{iff} \quad G\ f_c(m_4, m_5, t) = 1$

Proof: The block function of this property can be written as $f_c(m_4, m_5, t)$, where the output should hold *true* during the execution for any time t. To put it another way, the idea is to translate an LTL property into an invariant. As illustrated in Figure 5.13, the functionality of this block can be divided into three different functions that are denoted as $f_a(n,t)$, $f_b(input, value, t)$ and $f_c(m_4, m_5, t)$, where the output of each function based on the evolution of time t is as follows:

$$f_a(n,t) = \begin{cases} 1 & \text{if} \quad 0 \leq t \leq n \\ \\ 0 & \text{if} \quad t > n \end{cases}$$

$$f_b(input, value, t) = \begin{cases} 1 & \text{if} \quad input = value \\ \\ 0 & \text{if} \quad input \neq value \end{cases}$$

$$f_c(m_4, m_5, t) = \begin{cases} 0 & \text{if} \quad \neg m_4(t) \wedge \neg Pre^*(m_5, t) \wedge (t = n+1) \\ \\ 1 & \quad Otherwise \end{cases}$$

$Pre^*(m_5, t) = 1$ iff m_5 holds at least once before the current time.

By using fixed-step discrete sample time, at each time interval $t_n = nT_s + |T_o|$, Simulink executes all block outputs or block update methods based on the execution order denoted on the Figure 5.12. The sample time period T_s is always greater than zero and less than the simulation time T_{sim}. Offset time T_{sim} will be used if initial sample time delays is required. The number of periods (n) is also an integer that must satisfy: $0 \leq n \leq \frac{T_{sim}}{T_s}$. As a matter of fact, the simulation time is applicable when simulating the model, not property proving.

According to the code associated with *Within Implies* block in Appendix A, the initial output of m_6 is true. Table 5.1 denotes the output of each function and predecessor values over the evolution of time t for 8 time steps when $input = value$ at time step 6.
(Property falsified due to m_5 did not become true at least once within time $t \leq n$).

For simulation of the property $\varphi = \diamond_{[0,5]} p$, let us assume that the input signal has the values of $input = \{2, 4, 6, 8, 10, 12, 14\}$, and the desired value for the signal *input* is 8 in maximum 5 time steps (p must be true within 5 time steps). The result of simulation of the property as block outputs $M = \{m1, m2, m3, m4, m5, m6, m7\}$, by considering $T_{sim} = 7, T_s = 1, T_o = 0$ is listed in Table 5.2.

Table 5.1 Evolution of signal values over the time - $\diamond_{[0,5]}p$

t	0	1	2	3	4	5	6	7	8
m_4	1	1	1	1	1	1	0	0	0
m_5	0	0	0	0	0	0	1	0	0
m_6	1	1	1	1	1	1	0	1	1
$Pre^*(m_5)$	...	0	0	0	0	0	0	1	1

Table 5.2 Evolution of block outputs over the time - $\diamond_{[0,5]}p$

t	0	1	2	3	4	5	6	7
m_1	1	1	1	1	1	1	1	1
m_2	0	1	2	3	4	5	6	7
m_3	1	2	3	4	5	6	7	8
m_4	1	1	1	1	1	1	0	0
m_5	0	0	0	0	1	0	0	0
m_6	1	1	1	1	1	1	1	1
m_7	1	1	1	1	1	1	1	1

Property *p Until q*

This LTL property denoted by $p \cup q$ and holds, if p holds until q occurs, i.e., there is a state on the path at which q holds, and at every state before p holds.

Specifying this property by assertion which is supported by Simulink Design Verifier is not straightforward. Let us assume that p holds *true* in consecutive steps but still q is not true during the execution path. So if we make the output signal (*result* in Figure 5.14) based on assertion for the trueness of the p, the result will not be accurate because we don't know if the q will hold *true* in the next step or not.

The implementation of this property is illustrated in Figure 5.14. To produce this, we employed a Stateflow chart named 'Calculate' to make *enable* and *output* signals. The output of this block is connected to a *Proof Objective* block which verifies where the property can be satisfied or not. The Stateflow has two inputs p and q and one output as *prop*.

A discrete state $s \in S$ of a Simulink model with Stateflow chart consists of the set of *active states* in a Stateflow of the model along with the various choices of the conditional blocks in the model.
$S = \{main, main.s_1, main.s_2, main.s_3, main.s_4\}$ denotes the set of discrete states in the

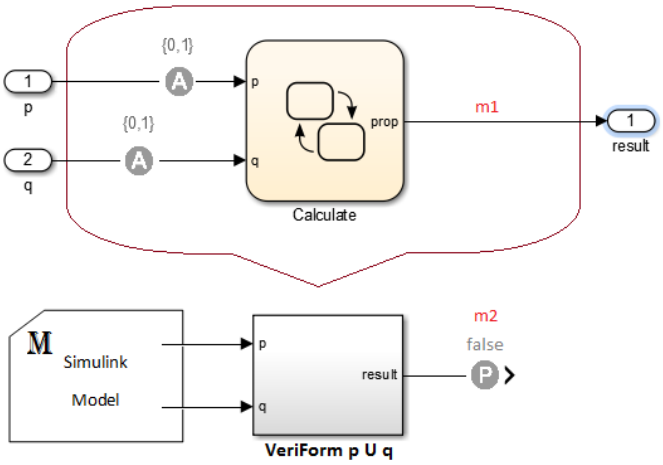


Figure 5.14 LTL $p\ U\ q$

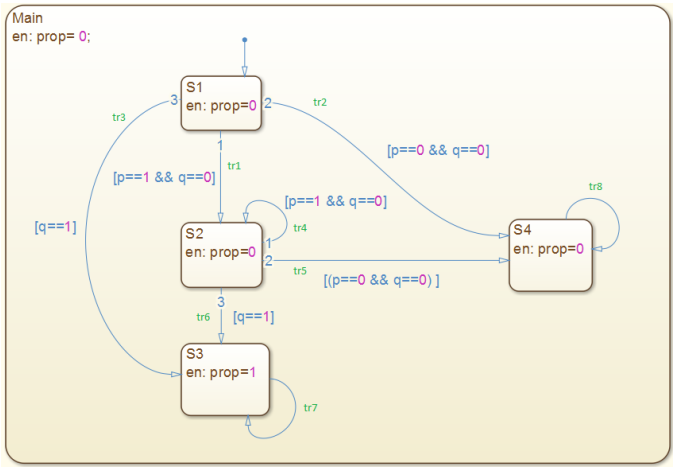


Figure 5.15 Stateflow in $p\ U\ q$

proposed Stateflow chart.

In the Stateflow, composition of states is either Parallel (And) or Exclusive (OR). In addition, A state definition sd is a triplet of actions $A = \{\mathbf{e}ntry, \mathbf{d}uring, e\mathbf{x}it\}$, executed respectively upon entering, during, and exiting the state, an internal composition, and a list of outgoing transitions [37]. As such, Outgoing transitions for the *Calculate* Stateflow chart as illustrated in Figure 5.15, is defined as: $TR = \{tr_1, tr_2, tr_3, tr_4, tr_5, tr_6, tr_7, tr_8\}$.

The state definition list SD which associates state definitions sd with corresponding states in the chart is denoted as:

$$SD = \{Main : sd_0; S1 : sd_1; S2 : sd_2; S3 : sd_3; S4 : sd_4\}$$

The definition of the states for the Stateflow depicted in Figure 5.15, can be listed as:

— $sd_0 = ((A.e, A.d), OR)$
— $sd_1 = ((A.e), OR, \{tr_1, tr_2, tr_3\})$
— $sd_2 = ((A.e), OR, \{tr_4, tr_5, tr_6\})$
— $sd_3 = ((A.e), OR, \{tr_7\})$
— $sd_4 = ((A.e), OR, \{tr_8\})$

As per shown in definition of state *SD*, chart has four sub-states. Since states has a OR composition, at each time step only the active state is evaluated depending on the inputs. Initially, in the Stateflow illustrated in Figure 5.15, S1 is active and $prop = 0$ and the goal is to have eventually $prop = 1$. The property is considered as satisfied as soon as state S3 is active. On the contrary the property considered as falsified if state S4 is active or state S3 has never reached. After the second step, if we have only $p \wedge \neg q$, the state S2 remains active and the state S3 never reaches, consequently the property becomes falsified. According to the Stateflow, if q holds in the first state, it reaches the state S3 and the property becomes satisfied.

Formalization: Let $\mathbb{M}$ be a Simulink model and p, q be two formulas:

$\mathbb{M} \models p\ U\ q \quad \text{iff} \quad F\ prop$

Proof: The block function of this property is denoted as $result = Calculate(p, q)$, where to declare that if the path π satisfies $F\ prop$, it means that the path allows to reach to state S3 in the Stateflow (Figure 5.15). According to the Stateflow, S3 is reached if all preceding states satisfy $p\ p\ p\ ...\ q$, which means that the path satisfies $p\ U\ q$. In addition, if the path satisfies $p\ U\ q$, then the Stateflow reaches the state S3.

Finally, m_3 as the output of this property, must *eventually* hold *true* during the entire execution of the chart.

In order to trace $p\ U\ q$, by specifying the time offset $T_o = 0$ and sample time period $T_s = 1\ sec$, the status of the states as s_a (*Active States*) and the output value *prop* of Stateflow for $n = 7$ time steps are:

In terms of evolution of time, the execution order *Proof Objective* block is after the 'Calculate' chart. Thus, in each time step of the execution, inputs of the Stateflow are evaluated first and then its outputs are updated and fed to *Proof Objective* block.

Table 5.3 Evolution of variables over the time - $p\ U\ q$

t	0	1	2	3	4	5	6	7
p	1	1	1	1	1	1	0	0
q	0	0	0	0	1	0	0	0
S_a	S_1	S_2	S_2	S_2	S_3	S_3	S_3	S_3
$prop$	0	0	0	0	1	1	1	1

Property $p\ Until_{[0,n]}\ q$

This LTL property denoted by $p \cup_{[0,n]} q$, holds for a given path π, if p holds until q occurs within n steps, and p must be satisfied before q. As described earlier, the only difference from implementation of $p \cup q$ is that the property should be satisfied during the n specified time steps on the path.

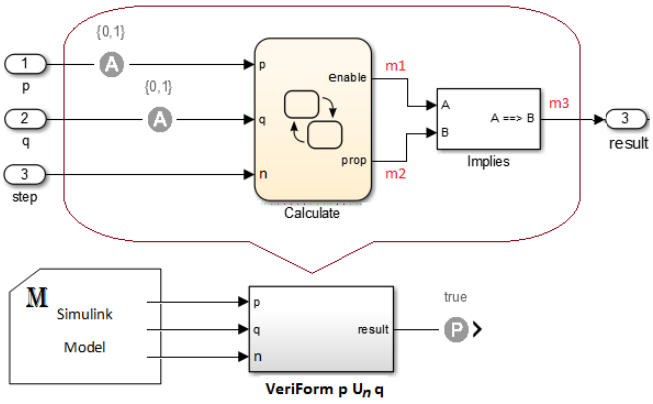


Figure 5.16 LTL $p\ U_{[0,n]}\ q$ Property

In order to specify this property, we used Stateflow to make *enable* and *prop* signals which are then sent to *Imply* block. The only difference to $p \cup q$ is that the property should be satisfied within n time steps of the path π. This time step limitation is implemented in the Stateflow by defining a local variable c that is initially set to zero.

This property comes with three inputs. Two inputs for p and q, and the third value t is the duration steps in the path that property should be satisfied.

The stateflow *Calculate*, has three inputs p, q, n, and two outputs *enable* and *prop*. The output *enable* is used to become true and remain true as soon as q became true after consecutive true occurrence of p in specified n time steps. It also becomes true and remains true even

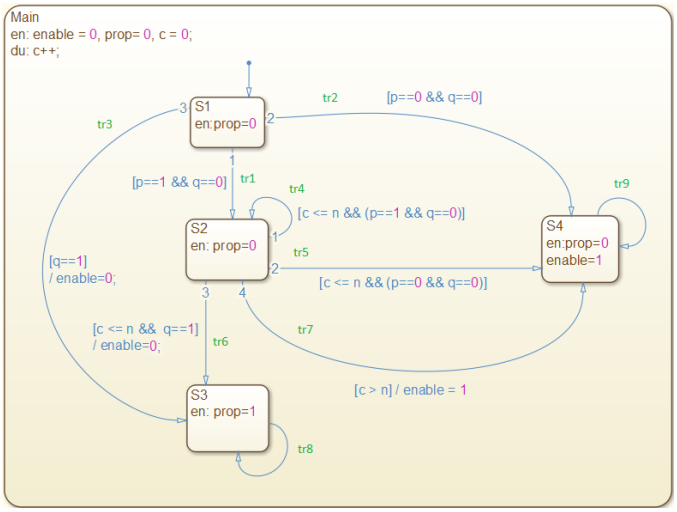


Figure 5.17 Stateflow in $p\ U_{[0,n]}\ q$

after q is not true after p (to set the *result* to false). In other words, this assures to set true for the first input of *Imply* block, otherwise the output of *Imply* block will be true for false inputs. The *prop* has a value false unless q becomes true in n time steps after consecutive true values of p.

Outgoing transitions for the *Calculate* Stateflow chart as illustrated in Figure 5.17, is defined as:
$TR = \{tr_1, tr_2, tr_3, tr_4, tr_5, tr_6, tr_7, tr_8, tr_9\}$.

The state definition list SD which associates state definitions sd with the corresponding states in the chart can be denoted as:

$$SD = \{Main : sd_0; S1 : sd_1; S2 : sd_2; S3 : sd_3; S4 : sd_4\}$$

The definition of the states for the Stateflow depicted in Figure 5.17, can be listed as:

- $sd_0 = ((A.e, A.d), OR)$
- $sd_1 = ((A.e), OR, \{tr_1, tr_2, tr_3\})$
- $sd_2 = ((A.e), OR, \{tr_4, tr_5, tr_6, tr_7\})$
- $sd_3 = ((A.e), OR, \{tr_8\})$
- $sd_4 = ((A.e), OR, \{tr_9\})$

Formalization: Let $\mathbb{M}$ be a Simulink model and p, q are two formulas:

$\mathbb{M} \models\ p\ U_{[0,n]}\ q \quad$ iff $\quad G\ enable \Rightarrow prop$

Proof: The block function of this property, denoted as $result = Calculate(p, q, n)$, where the output $result$ should hold *true* during the execution for all time steps t. The function $Calculate$ that is encased in this block function, reads p and q and produces two outputs based on the inputs p, q at each time steps t. The output of the block function $Calculate(p, q, n)$ based on the evolution of time $t \in \{0, ..., n\}$ is shown as:

$$prop_{(t)} = \begin{cases} 1 \quad \text{if} \quad [(c_{(t)} \leq step) \wedge \\ \qquad\quad (q_{(t)} \vee (q_{(t)} \wedge Pre(p, t)))] \vee \\ \qquad\quad [c_{(t)} > step \wedge Pre(prop, t)] \\ \\ 0 \quad \text{if} \quad \neg p_{(t)} \wedge \neg q_{(t)} \wedge c_{(t)} < step \end{cases}$$

$$enable_{(t)} = \begin{cases} 1 \quad \text{if} \quad [(c_{(t)} \leq step) \wedge \\ \qquad\quad (q_{(t)} \vee (q_{(t)} \wedge Pre(p, t)))] \vee \\ \qquad\quad [c_{(t)} \leq step \wedge \neg p_{(t)} \wedge \neg q_{(t)}] \vee \\ \qquad\quad (c_{(t)} = step \wedge \neg q_{(t)}) \ \vee (c_{(t)} > step) \\ \\ 0 \qquad otherwise \end{cases}$$

To clarify, c is the local variable defined in the Stateflow and acts as a step counter. Finally, m_3 as the output of this property, must hold *true* during the entire execution time $t \in \{0, ..., n\}$ time, and corresponds to the result of $m_1 \Rightarrow m_2$.

In order to trace $p\ U_{[0,5]}\ q$, by specifying the time offset $T_o = 0$ and sample time period $T_s = 1$, the status of the states as s_a (*Active States*) and the output values *enable* and *prop* of Stateflow for 7 time steps are shown in Table 5.4. As can be seen in Table 5.4, the property is not satisfied due to q became true at time step $t > 5$. Consequently, the last row of the table denotes that the property did not satisfy $G\ enable \Rightarrow prop$.

In terms of evolution of time, the execution order of *Implies* block is after 'Calculate' chart, and *Proof Objective* block is after the *Implies* block. Thus, in each time step of the execution, first inputs of the Stateflow are evaluated and and the outputs are updated and fed to *Implies* block.

5.6 Evaluation

Following this section, we introduce a case study in order to use the proposed blocks for LTL properties to specify some requirements of a home heating control system. The properties

Table 5.4 $p\ U_{[0,5]}\ q$ - Evolution of variables over the time

t	0	1	2	3	4	5	6	7
c	0	1	2	3	4	5	6	7
p	1	1	1	1	1	1	0	1
q	0	0	0	0	0	0	1	1
S_a	S_1	S_2	S_2	S_2	S_2	S_2	S_4	S_4
$enable$	0	0	0	0	0	0	1	1
$prop$	0	0	0	0	0	0	0	0
$enable \Rightarrow prop$	1	1	1	1	1	1	0	0

are specified based on the requirement specifications of the case study and the verification results are denoted right after. Since Simulink semantics depends on the simulation method, we restrain ourselves only to one method, namely, 'solver: *fixed-step, discrete*' and 'mode: *auto*'. This settings is also used for Simulink Design Verifier that has only support for discrete time systems. We assume that for every input of the model (i.e., every input of the controller) the sampling time is explicitly specified.

5.6.1 Case Study

In this section we present a case study as a Simulink model for the home heating control system. The model is depicted in Figure 5.18.

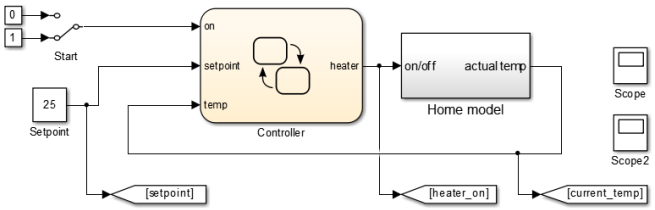


Figure 5.18 Home Heating System

The home heater control logic lies within the Stateflow chart and controls when the heater is turned on and off. Initially, the heater is turned off through the use of the *start_switch* block. When the system starts, the controller turns the heater on. During the system run, current temperature is also fed to controller and then the controller checks if the temperature is reached to the set-point and then it turns the heater off . If the set-point is reached the controller turns off the heater and waits until the temperature goes lower than the set-pint

and then turns it on to keep the home temperature around the set-point.

The sample time for the model of the case study is set to 1 and fixed-step discrete time is also used. The simulation of the case study for the set-point equal to 25 is shown in Figure 5.19.

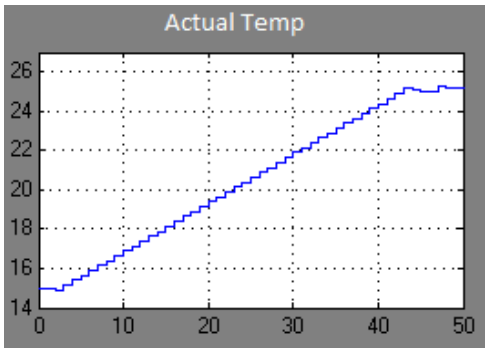


Figure 5.19 Simulation: Home Heating System

As can be seen in Figure 5.19, the home temperature reaches the set-point at time step 40.

Requirements

For the presented model consider the requirements listed below that we would like to verify:

1. The home temperature eventually reaches the set-point.
2. The home temperature eventually reaches the set-point in 45 time steps.
3. The heater is *on* until the home temperature reaches the set-point.
4. The heater is *on* until the home temperature reaches the set-point in 45 time steps.

Property Specification

In the specification the *Assumption* and the *Proof Objective* blocks (called as P-Block) are also used. With a proof objective, one can design a range of values that a specific signal has to hold during program execution. With an assumption, one can restrict input signals to a range of values for the analysis of the respective proof objective.

Requirement 1:
This property verifies if the room temperature eventually reaches the set-point specified at the beginning. Since the number of steps for verifying this property is unspecified, the Simulink Design Verifier can not tell directly that if the property holds or not. To solve this

by negation, the specified *P-Block* is set to *false* checks if the property doesn't hold during the verification process.

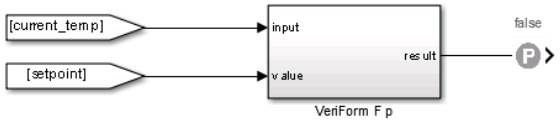


Figure 5.20 Requirement 1

Requirement 2:

For specifying this property, first we represent the number of time steps required, current_temp as input signal and the set-point as desired value. The Output goes to the *P-Block* which is a property block and as specified should hold *true* during the verification. The specification for this property is illustrated in Figure 5.21

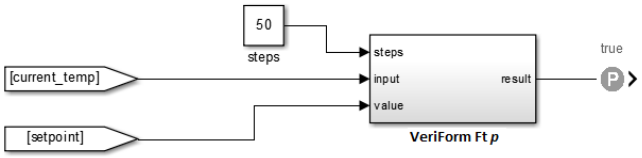


Figure 5.21 Requirement 2

Requirement 3:

In this property, when the controller was started, the heater should turned on and remain *on* until the temperature reaches the set-point or set-point plus 0.5 degree. For illustration, this requirement is broken down roughly into two pieces of interest:

— The current temperature is greater than or equal to the set-point.

— The current temperature is less than the set-point plus margin (0.5 degree).

The heater status is considered as first input (p) of the verification sub-system, and logical AND of the two above mentioned conditions considered as input (q). The output goes to the *P-Block* and as specified should hold *true* during the verification. The specification of this property is illustrated in Figure 5.22.

Requirement 4:

In this property, when the controller was started, the heater should turned on and remain *on* until the temperature reaches the set-point or set-point plus 0.5 degree in the specified time steps. The only difference between this requirement and the previous one is that the there

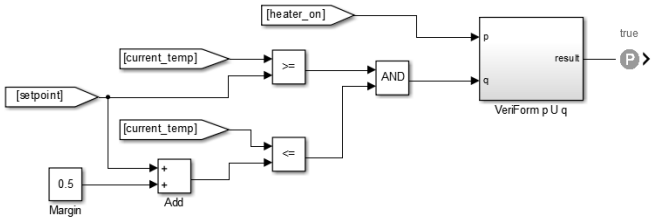


Figure 5.22 Requirement 3

is a limitation of time steps that are allowed until the property being considered as satisfied. The number of time steps can be set as a constant value to the third input of the verification sub-system. The specification of this property is illustrated in Figure 5.23.

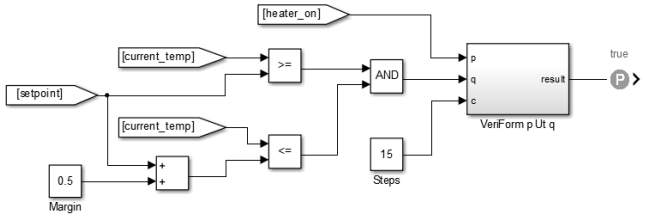


Figure 5.23 Requirement 4

5.6.2 Verification

Verification of the proposed properties are done on a computer having an Intel Core 2 Quad CPU with 6GB of RAM. We employed Matlab version 2013b and Simulink Design Verifier toolbox is also used as the verification engine.

As the simulation of the model shows in Figure 5.19, by specifying the set-point equal to 25, the home temperature reaches to the set-point at time step 44. Table 5.5 depicts the result of the property proving for different properties with different inputs.

5.7 Conclusion

Simulink and Stateflow have a suitable environment to model and simulate the embedded systems. We have outlined an approach to the verification of some linear temporal logic properties in Simulink. This is done on models through the use of Stateflow and some Simulink blocks from the standard and design verifier libraries. Although there is a lack of formal and rigorous semantics for the modeling language of this tool, we presented some

Table 5.5 Verification results

Row	**Property**	**SetPoint**	**Steps**	**Elapsed**	**Result**
1	R I	25	-	$2_m, 10_s$	Satisfied
2	R II	25	50	$3_m, 24_s$	Satisfied
3	R II	25	30	33_s	Falsified
4	R III	25 + 0.5	-	1_s	Satisfied
5	R III	25 + 0.2	-	3_s	Falsified
6	R IV	25	45	$2_h, 3_m, 24_s$	Satisfied
7	R IV	25	15	3_s	Falsified
8	R IV	25	25	3_s	Falsified
9	R IV	25	35	$30_m, 56_s$	Falsified

basic LTL properties and evaluated them as a case study using Simulink Design Verifier. The LTL2SL tool is also introduced to facilitate the instrumentation process of Simulink models with proposed customized properties. For the future work, we plan to define more complex LTL properties and add them to our customized library.

CHAPTER 6 GENERAL DISCUSSION

This chapter gives an overview of our work and contributions. It also presents a discussion of the approach followed, as well as an analysis of the outcomes.

The approach of focusing on the formal verification of software from the model, helps developers for building a system that functions dependably irrespective of its complicacy. To put it differently, using Model-Based Design in safety-related applications and using formal verification, supports preventing catastrophic consequences.

By the same token, the work presented in this thesis has debuted with a review of literature that is devoted to formal verification of Simulink models. At first, formal verification was briefly described and some verification tools were reviewed. The limitation of the block library provided by Simulink Design Verifier, that obliged other authors to apply the model transformation and employing different verification tools, is also described after. In the later chapters, we presented an approach for specifying and verifying the requirements of the system in an integrated environment from Mathworks.

6.1 Synthesis of work

In the first part, the research focuses on the modeling a concurrent system in Simulink, formalizing the requirement specifications and then verification of specified properties. To do this, a medical device known as Endotracheal intubation is modeled in Simulink, using the given Grafcet. The controller is constructed by Stateflow in Simulink. In order to make this possible to specify the properties, the values of required active states as well as active states corresponding to different components, are defined as output ports. Although the design of the system was provided through Grafcet, but implementation of a system with concurrent running components in Stateflow requires more considerations. In other words, when using a Stateflow in the model, it is advised to take this into account that Simulink Design Verifier does not support all Stateflow features. For example, attempts to call MATLAB functions or accessing to variables in the MATLAB workspace from the Stateflow, and calls to certain C math function is not supported by Simulink Design Verifier [5].

With this in mind, the controller chart designed as *Intubation* superstate, which includes different states corresponding to each component of the system. All these components are defined as Parallel (AND) states, so that in every moment of running the model, at least one substate has to be active in each state.

In order to hamper recursion and cyclic behavior in the chart, components interact each other through sending direct broadcast events. In addition, the current state and values of the processes are maintained using local and output variables. To put it differently, the *Active State Output Data* feature in the Stateflow employed to produce -1, 0 or 1 respectively correspond to the position of the cylinder at: *rear*, *center* and *front*. The temporal logic operator *after* is used to construct *Timer* states as well as the lag in the *Cylinder* state. Likewise, the *Detector* block from Temporal Operator library provided by Simulink Design Verifier is also employed to specify a time related property.

The second part is in pursuit of the first part of work. Moreover, this part involves the formalization of some safety and temporal requirements based on the events issued from the controller. Since the controller is designed in the Stateflow, working with events can be divided in two different categories. One is issuing the event from the Stateflow, and another is handling the issued event with other blocks in the same Simulink model. In fact, all temporal properties are designed and formalized for a bounded time slots that is specified by running duration or a procedure execution (e.g. when controller issues an inflate or deflate procedure).

In order to issue the event and activate other blocks in the same model, we used output events in the Stateflow. In fact, this type of event allows a chart to notify other blocks in a model about events that happen in the chart. We employed the triggered subsystem, to specify the first property which stands to verify the number of fill attempt issues by the controller to inflate a balloon. In particular, it has a single control input, called the trigger input that determines whether the subsystem executes. Moreover, the subsystem executes each time a trigger event occurs, and it outputs the number of events received so far to an Embedded-Matlab function. Finally, the output of this function is an invariant that can be verified by Simulink Design Verifier. In addition, the function block also has an input port which the value of the incoming signal bounds the time steps. To explain, whenever the input for this port is false, it tells to the function that the examining time is over. In the second and third property the authors again employs the Embedded-Matlab function corresponding to operator U in linear temporal logic. These functions also benefit from an input port which bound the number of time steps. Since the execution order of the function specified to be after other blocks in the model, its update method is called after. Finally, the output of this function will be the output of the property which can be verified by the Simulink Design Verifier.

In the third part of work, we propose a technique to facilitate formalizing some LTL properties so that they can be added to the Simulink model as some customizable blocks. In addition,

the subsystem advantage is also used to group and encapsulation the specified properties and hosted in the *VeriForm* library. We also present how the Simulink model can be instrumented by using the proposed custom library and show the way that the functionality is formalized. In particular, to specify LTL property, the idea is to translate the selected LTL properties to an invariant so that they can be verified by the Simulink Design verifier.

To specify the LTL property $\Diamond_{[0,n]}p$, we used the standard Simulink library to build a required amount of time steps n, and a temporal operator *Within Implies* which is provided by Simulink Design Verifier library. Moreover, in the implementation the *Unit Delay* block is also used to prevent the algebraic loop. The output of property is an invariant and must hold true at any time step in a given path π.

Specifying the property $p \cup q$ with an assertion in Simulink is also not straightforward. To implement this property, authors employed a Stateflow called *Calculate*, which hast two input corresponding to p and q and one output that must hold true at least once within the execution. Similarly, the property $p \cup_{[0,n]} q$, is specified using the Stateflow which has two outputs that fed to an *Implies* block. The output of the *Implies* block must hold true for all time steps to satisfy the property.

There are two different ways that the user can employ proposed *VeriForm* library: 1) Using this predefined library while creating the Simulink model, 2) Using LTL2SL tool that can add the chosen block to the previously made Simulink model. Since applying above mentioned properties, might be difficult for less-expert users, the LTL2SL tool is also proposed to facilitate the process.

6.2 Analysis of the achievements

Apart from the complexity of the requirement specification, defining appropriate inputs is also important. In other words, the assumption of signal inputs in the model, is one of the significant factors that can have direct impact on the verification time of specified properties.

By optimizing the Controller chart in the model described in Chapter 3 and 4, we obtained the shorter verification time. In particular, specifying the boundaries for local variables and output signals as well as using the *Proof Assumption* block with appropriate values, are part of our optimization.

On the other hand, choosing a proper transition order as well as the suitable guard for the transition conditions in the Stateflow can increase the performance. Moreover, there is a parameter in the configuration of Simulink Design Verifier pertaining to the maximum verification time. As such, using the optimized model with specifying appropriate signal values

helped us to prevent having verification results as *undecided*. On the contrary, choosing the wrong values results the model becomes contradictory in its configuration, and all objectives become Unsatisfiable/Falsifiable.

CHAPTER 7 CONCLUSION AND RECOMMENDATIONS

Software plays increasingly a substantial role in embedded systems used particularly for healthcare, automotive and avionics. On the other hand, most of the application induced failures are due to a failure in design phase of the software. In addition, activities related to verification and validation are becoming huge and quite costly when the complexity and size of the systems grows. As such, design of safety-critical systems requires to benefit from a more systematic development process. By the same token, one of the ways to achieve this goal consists of using formal verification tools such as Simulink Design Verifier. In this thesis, we have addressed the problem of specification and verification of LTL properties in the context of Simulink Design Verifier. We have shown how to combine Simulink blocks so as to translate LTL properties into equivalent invariants. In this chapter, we present a summary of the contributions which have been made, followed by a critique of our work through their limitations. Finally, we state the avenues on which axes will be our future work.

7.1 Summary of work

Our research has given rise to three main contributions. The first contribution consists of a case study that allows us to study, apply and evaluate the power and features of Simulink development environment. A concurrent system corresponding to a medical device known as endotracheal intubation, is modeled in Simulink. Since the specifications of the control unit of the mentioned device is given in Grafcet, authors needed to explore the way to construct its Simulink model. We used formal approach and model-based design in order to specify and formally verify the functionalities of this system. The next step was formalizing the requirement specifications and then verification of specified safety properties. In order to prevent recursion and cyclic behavior in the chart, components are designed to interact each other through sending direct broadcast events. Moreover, event passing/handling and synchronization is efficiently provided. Finally, various proof strategies along with the appropriate set of data for this model are presented in order to prove the correctness of the model.

The contribution deals with the improvement and optimisation of the model of endotracheal intubation device, that allows a significant gain in the verification time of properties. Moreover, these improvement involves the formalization of some safety and temporal requirements based on the events issued from the controller. In the proposed solution for specifying the temporal properties, the triggered subsystem block as well as the Embedded-Matlab function

block from the standard Simulink library are employed. In other words, the Stateflow chart for the controller is modified to trigger an output event, and triggered subsystem block has the duty to capture the events. The rest of the works is handled through the Embedded-Matlab function block which makes an output as invariant. The output then is connected to P-Block to be verified by the Simulink Design Verifier. Furthermore, all temporal properties are designed and formalized for a bounded time slots which can be specified in the proposed property block.

Since the block library provided Simulink Design Verifier is limited to specify the safety properties, we proposed a technique to facilitate formalizing some LTL properties. In this case, we offer different customizable Simulink blocks corresponding to each property. In other words, the subsystem advantage is used to group the definition of specified properties. The standard Simulink blocks and the Stateflow chart are used to implement the specification for blocks corresponding to $\Diamond p$, $\Diamond_{[0,n]} p$, $p \cup q$ and $p \cup_{[0,n]} q$.

In particular, to specify LTL property, the idea was to translate the selected LTL properties to an invariant so that they can be verified by Simulink Design verifier. By the same token, LTL2SL tool is also proposed to facilitate the process of specifying the LTL properties for the less-expert users.

7.2 Limitations of the proposed solution

We have shown how to specify some basic linear temporal properties in Simulink models by offering some customizable blocks. In particular, the proposed blocks are limited to bounded time steps, and there is also no support for recursive LTL properties. However, further consideration is needed in order to improve the techniques to bring support for specifying more complex LTL properties. Although we have proposed LTL2SL tool in order to facilitate the process of instrumentation of the given model, it has limited to the first layer of model blocks. In other words, it does not support to automatically add proposed blocks inside the subsystems which are under the first layer in the model.

7.3 Future Work

Some techniques and directions for future work have already been mentioned in individual chapters of this thesis. Since recursive LTL properties are not included in the offered Simulink blocks, it is remarkable to have support for them in the future effort. Additionally, we will also investigate how to specify recursive LTL properties using Simulink Design Verifier. In the LTL2SL tool we considered to modify the given Simulink model and insert the property

block at top level in the model. It is interesting to improve the tool that can support inserting blocks into subsystems at different level of the model.

REFERENCES

[1] Intensive care hotline, intubation, http://intensivecarehotline.com/intubation/.

[2] Prover plug-in, prover, http://www.prover.com/products/.

[3] Scade, prover, http://www.esterel-technologies.com/products/scade-suite/.

[4] Simulink coder, the mathworks, http://www.mathworks.com/products/simulink-coder/.

[5] Simulink design verifier, http://www.mathworks.com/products/sldesignverifier/.

[6] Simulink, the mathworks, http://www.mathworks.com/products/simulink/.

[7] Sysml forum: Sysml specification 'draft' 2005, http://www.sysml.org/.

[8] Tracheal intubation, https://en.wikipedia.org/wiki/tracheal_intubation.

[9] User's guide, the mathworks, http://www.mathworks.com/products/.

[10] V.S. Alagar and K. Periyasamy. *Specification of software systems*. Springer-Verlag New York Inc, 2011.

[11] Shahid Ali, Muhammad Sulyman, Mattias Nyberg, Jonas Westman, Giuseppe Dellapenna, Guillermo Rodríguez-Navas González, and Paul Pettersson. Applying model checking for verifying the functional requirements of a scania's vehicle control system. *School of Innovation, Design and Engineering Malardalen University, Vasteras, Sweden*, 2012.

[12] Jirí Barnat, Petr Bauch, and Vojtech Havel. Temporal verification of simulink diagrams. In *High-Assurance Systems Engineering (HASE), 2014 IEEE 15th International Symposium on*, pages 81–88. IEEE, 2014.

[13] A. Behboodian. Model-based design. *DSP Magazine, May*, 2006.

[14] M Beine. A model-based reference workflow for the development of safety-critical software. *Embedded Real Time Software and Systems*, pages 1–6, 2010.

[15] Olof Bergquist and Marcus Sjödin. Modeling and verification of a stepper motor supervisory controller. 2008.

[16] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without bdds. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 193–207, 1999.

[17] A. Biere, A. Cimatti, E.M. Clarke, O. Strichman, and Y. Zhu. Bounded model checking. *Advances in computers*, 58:117–148, 2003.

[18] P. Bjesse. What is formal verification? *ACM SIGDA Newsletter*, 35(24):1, 2005.

[19] Olivier Bouissou and Alexandre Chapoutot. An operational semantics for simulink's simulation engine. *ACM SIGPLAN Notices*, 47(5):129–138, 2012.

[20] R.E. Bryant. Graph-based algorithms for boolean function manipulation. *Computers, IEEE Transactions on*, 100(8):677–691, 1986.

[21] Alexandre Chapoutot and Matthieu Martel. Abstract simulation: a static analysis of simulink models. In *Embedded Software and Systems, 2009. ICESS'09. International Conference on*, pages 83–92. IEEE, 2009.

[22] Chunqing Chen. Tic library functions for simulink library blocks.

[23] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In *Computer Aided Verification*, pages 359–364. Springer, 2002.

[24] E. Clarke. Model checking. In *Foundations of software technology and theoretical computer science*, pages 54–56. Springer, 1997.

[25] E. Clarke and E. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. *Logics of Programs*, pages 52–71, 1982.

[26] E. Clarke, K. McMillan, S. Campos, and V. Hartonas-Garmhausen. Symbolic model checking. In *Computer Aided Verification*, pages 419–422. Springer, 1996.

[27] E.M. Clarke and J.M. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys (CSUR)*, 28(4):626–643, 1996.

[28] E. Denney. A software safety certification for automated code generators. 2006.

[29] E. Denney and S. Trac. A software safety certification tool for automatically generated guidance, navigation and control code. In *Aerospace Conference, 2008 IEEE*, pages 1–11. IEEE, 2008.

[30] V. D'Silva, D. Kroening, and G. Weissenbacher. A survey of automated techniques for formal software verification. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 27(7):1165–1178, 2008.

[31] E Allen Emerson. Temporal and modal logic. *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, 995:1072, 1990.

[32] J.F. Etienne, S. Fechter, and E. Juppeaux. Using simulink design verifier for proving behavioral properties on a complex safety critical system in the ground transportation domain. *Complex Systems Design & Management*, pages 61–72, 2010.

[33] I. Fey, J. Mller, and M. Conrad. Model-based design for safety-related applications. *Proceedings of SAE Convergence*, 2008.

[34] M. Fowler. *UML distilled: a brief guide to the standard object modeling language*. Addison-Wesley Professional, 2004.

[35] E. Gamma. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[36] Nicholas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data flow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[37] G. Hamon and J. Rushby. An operational semantics for stateflow. *Fundamental Approaches to Software Engineering*, pages 229–243, 2004.

[38] D. Harel. Statecharts: A visual formalism for complex systems. *Science of computer programming*, 8(3):231–274, 1987.

[39] X. He, Z. Ma, W. Shao, and G. Li. A metamodel for the notation of graphical modeling languages. In *Computer Software and Applications Conference, 2007. COMPSAC 2007. 31st Annual International*, volume 1, pages 219–224. IEEE, 2007.

[40] Christian Heinzemann, Jan Rieke, Jana Bröggelwirth, Andrey Pines, and Andreas Volk. Translating mechatronicuml models to matlab/simulink and stateflow. *Software Engineering Group, University of Paderborn, Tech. Rep. tr-ri-13-330*, 2013.

[41] E. Iee. Ieee standard glossary of software engineering terminology. 1990.

[42] Zhihao Jiang, Miroslav Pajic, Allison Connolly, Sanjay Dixit, and Rahul Mangharam. Real-time heart model for implantable cardiac device validation and verification. In *Real-Time Systems (ECRTS), 2010 22nd Euromicro Conference on*, pages 239–248. IEEE, 2010.

[43] LA Johnson. Do-178b: Software considerations in airborne systems and equipment certification. *Crosstalk, The Journal of Defense Software Engineering*, 11(10), 1998.

[44] S. Katz. *Techniques for increasing coverage of formal verification*. PhD thesis, The Technion-Israel Institute of Technology, 2001.

[45] Konstantin Korovin. Chapter 14 - linear temporal logic. *http://www.control.aau.dk/ raf/hybrid/korovinkchapter14.pdf*.

[46] Jerry Krasner. Model-based design and beyond: Solutions for todays embedded systems requirements. Technical report, Mathworks, 2004.

[47] Heiko Krumm. Temporal logic. Technical report, Technical report, Department of Computer Science, University of Dortmund, 2000.

[48] Lj Lazić and D Velašević. Applying simulation and design of experiments to the embedded software testing process. *Software Testing, Verification and Reliability*, 14(4):257–282, 2004.

[49] Florian Leitner. Evaluation of the matlab simulink design verifier versus the model checker spin. 2008.

[50] Florian Leitner and Stefan Leue. Simulink design verifier vs. spin a comparative case study. In *Proceedings of FMICS*, 2008.

[51] Marcus Liliegard and Viktor Nilsson. Model-based testing with simulink design verifier.

[52] Karthikeyan Manamcheri Sukumar. Translation of simulink-stateflow models to hybrid automata. 2011.

[53] B Meenakshi, Abhishek Bhatnagar, and Sudeepa Roy. Tool for translating simulink models into input language of a model checker. In *Formal Methods and Software Engineering*, pages 606–620. Springer, 2006.

[54] Jon Friedman Mike Anthony. Model-based design for large safety-critical systems: A discussion regarding model architecture. Technical report, Mathworks.

[55] S. Miller, E. Anderson, L. Wagner, M. Whalen, and M. Heimdahl. Formal verification of flight critical software. In *Proceedings of the AIAA Guidance, Navigation and Control Conference and Exhibit*, pages 15–18, 2005.

[56] B. Murphy, A. Wakefield, and J. Friedman. Best practices for verification, validation, and test in model-based design. *The Mathworks, Inc*, 2008.

[57] Anitha Murugesan, Michael W Whalen, Sanjai Rayadurgam, and Mats PE Heimdahl. Compositional verification of a medical device system. In *Proceedings of the 2013 ACM SIGAda annual conference on High integrity language technology*, pages 51–64. ACM, 2013.

[58] Miroslav Pajic, Insup Lee, Rahul Mangharam, and Oleg Sokolsky. Upp2sf: Translating uppaal models to simulink. *University of Pennsylvania, Tech. Rep*, 2011.

[59] Paula J Pingree, Erich Mikk, Gerard J Holzmann, Margaret H Smith, and Dennis Dams. Validation of mission critical software design and implementation using model checking [spacecraft]. In *Digital Avionics Systems Conference, 2002. Proceedings. The 21st*, volume 1, pages 6A4–1. IEEE, 2002.

[60] K. Popovici and M. Lalo. Formal model and code verification in model-based design. In *Circuits and Systems and TAISA Conference, 2009. NEWCAS-TAISA'09. Joint IEEE North-East Workshop on*, pages 1–4. IEEE, 2009.

[61] Paulo Portugal and Adriano Carvalho. The grafcet specification. 2005.

[62] R.S. Pressman. *Software Engineering-A Practitioners Approach-Required*. McGraw Hill, 1992.

[63] J.P. Queille and J. Sifakis. Specification and verification of concurrent systems in cesar. In *International Symposium on Programming*, pages 337–351. Springer, 1982.

[64] Yelamuri Srinivasa Rao. C-code generation from simulink models using flex and bison. *http://sourceforge.net/projects/sim2c/files/*.

[65] Pritam Roy and Natarajan Shankar. Simcheck: a contract type system for simulink. *Innovations in Systems and Software Engineering*, 7(2):73–83, 2011.

[66] J. Rushby. *Formal methods and the certification of critical systems*. SRI International, Computer Science Laboratory, 1993.

[67] Paul Seigman. *APPLYING THE SPIN MODEL-CHECKER TO MODEL-BASED DESIGNS IMPLEMENTED IN MATLAB SIMULINK/STATEFLOW.* 2006.

[68] Mary Sheeran and Gunnar Stålmarck. A tutorial on stålmarck's proof procedure for propositional logic. *Formal Methods in System Design*, 16(1):23–58, 2000.

[69] Neil R Storey. *Safety critical computer systems*. Addison-Wesley Longman Publishing Co., Inc., 1996.

[70] Paulo Tabuada and George J Pappas. Linear time logic control of discrete-time linear systems. *Automatic Control, IEEE Transactions on*, 51(12):1862–1877, 2006.

[71] A. Tiwari. Formal semantics and analysis methods for simulink stateflow models. *Unpublished report, SRI International*, 2002.

[72] Stavros Tripakis, Christos Sofronis, Paul Caspi, and Adrian Curic. Translating discrete-time simulink to lustre. *ACM Transactions on Embedded Computing Systems (TECS)*, 4(4):779–818, 2005.

[73] Richard Zurawski, Paulo Portugal, and Adriano Carvalho. The grafcet specification language. In *The Industrial Information Technology Handbook*, pages 993–1013. CRC Press, 2004.

APPENDIX A Within Implies Code

Function: Within Implies block code

```
1:  function out = whithin(u, reset, r)
2:  {
3:    if isempty(inU) then
4:        inU = false;
5:    endif
6:
7:    if isempty(rOcc) then
8:        rOcc = false;
9:    endif
10:
11:   if u && not(reset)
12:       inU = true;
13:       if r then
14:           rOcc = true;
15:       endif
16:       out = true;
17:   elseif not(u) && inU
18:       if not(rOcc) then
19:           out = false;
20:       else
21:           out = true;
22:       endif
23:       inU = false;
24:       rOcc = false;
25:   elseif
26:       out = true;
27:       rOcc = false;
28:   endif
29: }
```

APPENDIX B Embedded Matlab Function

Function: Embedded MATLAB Function 2

```
1:  function out = whithin(p, q, act)
2:  {
3:    persistent pre, done, res;
4:    if isempty(pre) then
5:        pre = false;
6:    endif
7:
8:    if isempty(done) then
9:        done = false;
10:   endif
11:
12:   if isempty(res) then
13:       res = false;
14:   endif
15:
16:   if act && not(done)
17:       if not(pre) then
18:           if q then
19:               done = true;
20:           elseif p && not(q) then
21:               pre = true;
22:           endif
23:           res = true;
24:       endif
25:
26:       if p && pre then
27:           res = true;
28:       endif
29:
30:       if not(p) && pre then
```

```
31:                 if not(q) then
32:                     res = false;
33:                 else
34:                     res = true;
35:                     done = true;
36:                 endif
37:             endif
38:
39:         elseif act && done
40:             res = true;
41:         elseif not(act) && pre then
42:             if done then
43:                 res = true;
44:             elseif q then
45:                 res = true;
46:             else
47:                 res = false;
48:             endif
49:         elseif not(act) && not(pre) then
50:             res = false;
51:         endif
52:         out = res;
53:     }
```